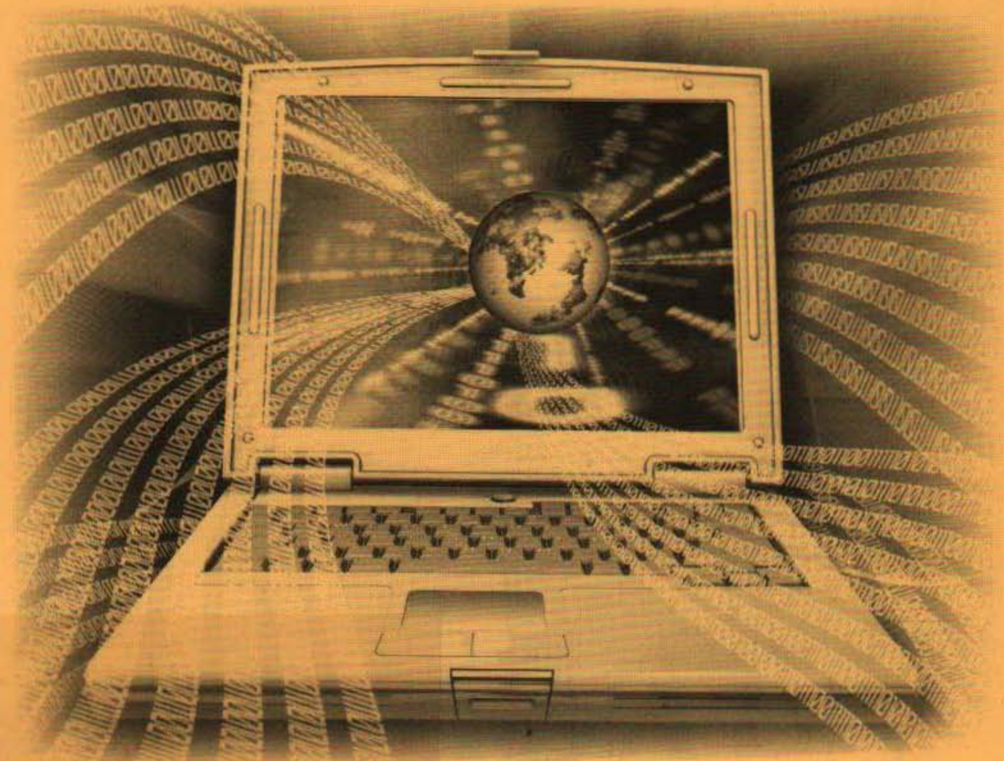
# FUNDAMENTOS DE PROGRAMACIÓN

José A. Cerrada Somolinos
Manuel E. Collado Machuca

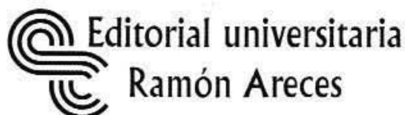


Editorial universitaria Ramón Areces

UNED

JOSÉ A. CERRADA SOMOLINOS
Catedrático de Lenguajes y Sistemas Informáticos (UNED)
MANUEL E. COLLADO MACHUCA
Catedrático de Lenguajes y Sistemas Informáticos (UPM)

# FUNDAMENTOS DE PROGRAMACIÓN



# Índice


**1 Introducción** . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

1.1 Máquinas y programas . . . . . . . . . . . . . . . . . . . . . 1

1.1.1 Máquinas programables . . . . . . . . . . . . . . . . . . 1

1.1.2 Concepto de cómputo . . . . . . . . . . . . . . . . . . . 3

1.1.3 Concepto de computador . . . . . . . . . . . . . . . . . . 4

1.2 Programación e ingeniería de software . . . . . . . . . . . . . 5

1.2.1 Programación . . . . . . . . . . . . . . . . . . . . . . . 5

1.2.2 Objetivos de la programación . . . . . . . . . . . . . . . 6

1.3 Lenguajes de programación . . . . . . . . . . . . . . . . . . . 7

1.4 Compiladores e Intérpretes . . . . . . . . . . . . . . . . . . . 9

1.5 Modelos abstractos de cómputo . . . . . . . . . . . . . . . . . 11

1.5.1 Modelo funcional . . . . . . . . . . . . . . . . . . . . . . 12

1.5.2 Modelo de flujo de datos . . . . . . . . . . . . . . . . . . 13

1.5.3 Modelo de programación lógica . . . . . . . . . . . . . . . 15

1.5.4 Modelo imperativo . . . . . . . . . . . . . . . . . . . . . 16

1.6 Elementos de la programación imperativa . . . . . . . . . . . . 19

1.6.1 Procesador, entorno, acciones . . . . . . . . . . . . . . . 19

1.6.2 Acciones primitivas. Acciones compuestas . . . . . . . . . 20

1.6.3 Esquemas de acciones . . . . . . . . . . . . . . . . . . . 21

1.7 Evolución de la programación . . . . . . . . . . . . . . . . . . 21

1.7.1 Evolución comparativa Hardware/Software . . . . . . . . 21

1.7.2 Necesidad de metodología y buenas prácticas . . . . . . . 22

**2 Elementos básicos de programación** . . . . . . . . . . . . . . . **25**

2.1 Lenguaje **C±** . . . . . . . . . . . . . . . . . . . . . . . . . . 25

2.2 Notación BNF . . . . . . . . . . . . . . . . . . . . . . . . . . 26

2.3 Valores y tipos . . . . . . . . . . . . . . . . . . . . . . . . . . 27

2.4 Representación de valores constantes . . . . . . . . . . . . . . 28

2.4.1 Valores numéricos enteros . . . . . . . . . . . . . . . . . 29

2.4.2 Valores numéricos reales . . . . . . . . . . . . . . . . . . 30

2.4.3 Caracteres . . . 31
2.4.4 Cadenas de caracteres (*strings*) . . . 31
2.5 Tipos predefinidos . . . 32
2.5.1 El tipo entero (`int`) . . . 33
2.5.2 El tipo real (`float`) . . . 35
2.5.3 El tipo carácter (`char`) . . . 37
2.6 Expresiones aritméticas . . . 39
2.7 Operaciones de escritura simples . . . 42
2.7.1 El procedimiento `printf` . . . 42
2.8 Estructura de un programa completo . . . 45
2.8.1 Uso de comentarios . . . 46
2.8.2 Descripción formal de la estructura de un programa . . . 46
2.9 Ejemplos de programas . . . 47
2.9.1 Escribir una fecha . . . 47
2.9.2 Suma de números consecutivos . . . 48
2.9.3 Área y volumen de un cilindro . . . 49

**3 Constantes y Variables . . . 51**
3.1 Identificadores . . . 51
3.2 El vocabulario de **C±** . . . 53
3.3 Constantes . . . 54
3.3.1 Concepto de constante . . . 54
3.3.2 Declaración de constantes con nombre . . . 55
3.4 Variables . . . 56
3.4.1 Concepto de variable . . . 56
3.4.2 Declaración de variables . . . 58
3.4.3 Uso de variables. Inicialización . . . 59
3.5 Sentencia de asignación . . . 61
3.5.1 Sentencias de autoincremento y autodecremento . . . 62
3.5.2 Compatibilidad de tipos . . . 62
3.6 Operaciones de lectura simple . . . 64
3.6.1 El procedimiento `scanf` . . . 65
3.6.2 Lectura interactiva . . . 66
3.7 Estructura de un programa con declaraciones . . . 68
3.8 Ejemplos de programas . . . 68
3.8.1 Ejemplo: Conversión a horas, minutos y segundos . . . 68
3.8.2 Ejemplo: Área y volumen de un cilindro . . . 69
3.8.3 Ejemplo: Realización de un recibo . . . 70

**4 Metodología de Desarrollo de Programas (I) . . . 73**
4.1 La programación como resolución de problemas . . . 73

4.2 Descomposición en subproblemas . . . . . 74
4.3 Desarrollo por refinamientos sucesivos . . . . . 76
4.3.1 Desarrollo de un esquema secuencial . . . . . 76
4.3.2 Ejemplo: Imprimir la silueta de una silla . . . . . 79
4.4 Aspectos de estilo . . . . . 80
4.4.1 Encolumnado . . . . . 80
4.4.2 Comentarios. Documentación del refinamiento . . . . . 81
4.4.3 Elección de nombres . . . . . 85
4.4.4 Uso de letras mayúsculas y minúsculas . . . . . 88
4.4.5 Constantes con nombre . . . . . 89
4.5 Ejemplos de programas . . . . . 90
4.5.1 Ejemplo: Imprimir la figura de un árbol de navidad . . . 90
4.5.2 Ejemplo: Calcular el costo de las baldosas . . . . . 92
4.5.3 Ejemplo: Calcular los días entre dos fechas . . . . . 93

**5 Estructuras Básicas de la Programación Imperativa . . . . . 97**
5.1 Programación estructurada . . . . . 97
5.1.1 Representación de la estructura de un programa . . . . . 98
5.1.2 Secuencia . . . . . 99
5.1.3 Selección . . . . . 100
5.1.4 Iteración . . . . . 100
5.1.5 Estructuras anidadas . . . . . 101
5.2 Expresiones condicionales . . . . . 101
5.3 Estructuras básicas en **C±** . . . . . 106
5.3.1 Secuencia . . . . . 106
5.3.2 Sentencia IF . . . . . 106
5.3.3 Sentencia WHILE . . . . . 109
5.3.4 Sentencia FOR . . . . . 110
5.4 Ejemplos de programas . . . . . 112
5.4.1 Ejemplo: Ordenar tres datos . . . . . 112
5.4.2 Ejemplo: Escribir un triángulo de dígitos . . . . . 114
5.4.3 Ejemplo: Elaboración de tickets y resúmenes . . . . . 116

**6 Metodología de Desarrollo de Programas (II) . . . . . 119**
6.1 Desarrollo con esquemas de selección e iteración . . . . . 119
6.1.1 Esquema de selección . . . . . 120
6.1.2 Esquema de iteración . . . . . 121
6.2 Ejemplos de desarrollo con esquemas . . . . . 123
6.2.1 Ejemplo: Imprimir el contorno de un triángulo . . . . . 123
6.2.2 Ejemplo: Imprimir el triángulo de Floyd . . . . . 129
6.3 Verificación de programas . . . . . 135

6.3.1 Notación lógico-matemática . . . . . . . . . . . . . . . . . . . 136
6.3.2 Corrección parcial y total . . . . . . . . . . . . . . . . . . . 136
6.3.3 Razonamiento sobre sentencias de asignación . . . . . . . . 137
6.3.4 Razonamiento sobre el esquema de selección . . . . . . . . 138
6.3.5 Razonamiento sobre el esquema de iteración: invariante, terminación. . . . . . . . . . . . . . . . . . . . . . . . . . . 140
6.4 Eficiencia de programas. Complejidad . . . . . . . . . . . . . . . . . 142
6.4.1 Medidas de eficiencia . . . . . . . . . . . . . . . . . . . . . . 142
6.4.2 Análisis de programas . . . . . . . . . . . . . . . . . . . . . . 143
6.4.3 Crecimiento asintótico . . . . . . . . . . . . . . . . . . . . . . 145

**Ejercicios sin resolver - I . . . . . . . . . . . . . . . . . . . . . . . . . 147**

**7 Funciones y Procedimientos . . . . . . . . . . . . . . . . . . . . . 151**
7.1 Concepto de subprograma . . . . . . . . . . . . . . . . . . . . . . . 151
7.2 Funciones . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
7.2.1 Definición de funciones . . . . . . . . . . . . . . . . . . . . . 154
7.2.2 Uso de funciones . . . . . . . . . . . . . . . . . . . . . . . . . 156
7.2.3 Funciones predefinidas . . . . . . . . . . . . . . . . . . . . . . 157
7.2.4 Funciones estándar . . . . . . . . . . . . . . . . . . . . . . . . 158
7.3 Procedimientos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
7.3.1 Definición de procedimientos . . . . . . . . . . . . . . . . . 160
7.3.2 Uso de procedimientos . . . . . . . . . . . . . . . . . . . . . . 161
7.3.3 Procedimientos estándar . . . . . . . . . . . . . . . . . . . . 162
7.4 Paso de argumentos . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
7.4.1 Paso de argumentos por valor . . . . . . . . . . . . . . . . . 162
7.4.2 Paso de argumentos por referencia . . . . . . . . . . . . . . 163
7.5 Visibilidad. Estructura de bloques . . . . . . . . . . . . . . . . . . . 165
7.6 Recursividad de subprogramas . . . . . . . . . . . . . . . . . . . . . 167
7.7 Problemas en el uso de subprogramas . . . . . . . . . . . . . . . . . 168
7.7.1 Uso de variables globales. Efectos secundarios . . . . . . 169
7.7.2 Redefinición de elementos . . . . . . . . . . . . . . . . . . . . 171
7.7.3 Doble referencia . . . . . . . . . . . . . . . . . . . . . . . . . 173
7.8 Ejemplos de programas . . . . . . . . . . . . . . . . . . . . . . . . . 174
7.8.1 Ejemplo: Raíces de una ecuación de segundo grado . . . 174
7.8.2 Ejemplo: Ordenar tres valores . . . . . . . . . . . . . . . . . 176
7.8.3 Ejemplo: Perímetro de un triángulo . . . . . . . . . . . . . 178

**8 Metodología de Desarrollo de Programas (III) . . . . . . . . . 181**
8.1 Operaciones abstractas . . . . . . . . . . . . . . . . . . . . . . . . . 181
8.1.1 Especificación y realización . . . . . . . . . . . . . . . . . . 182

8.1.2 Funciones. Argumentos . . . . . . . . . . . . . . . . . . . . 184
8.1.3 Acciones abstractas. Procedimientos . . . . . . . . . . . . . 186
8.2 Desarrollo usando abstracciones . . . . . . . . . . . . . . . . . . 188
8.2.1 Desarrollo descendente . . . . . . . . . . . . . . . . . . . . 188
8.2.2 Ejemplo: Imprimir la figura de un árbol de navidad . . . 189
8.2.3 Ejemplo: Imprimir una tabla de números primos . . . . . 196
8.2.4 Reutilización . . . . . . . . . . . . . . . . . . . . . . . . . . 199
8.2.5 Ejemplo: Tabular la serie de Fibonacci . . . . . . . . . . 200
8.2.6 Desarrollo para reutilización . . . . . . . . . . . . . . . . . 203
8.2.7 Desarrollo ascendente . . . . . . . . . . . . . . . . . . . . . 206
8.3 Programas robustos . . . . . . . . . . . . . . . . . . . . . . . . . 210
8.3.1 Programación a la defensiva . . . . . . . . . . . . . . . . . 210
8.3.2 Tratamiento de excepciones . . . . . . . . . . . . . . . . . . 213

**9 Definición de tipos . . . . . . . . . . . . . . . . . . . . . . . . 219**
9.1 Tipos definidos . . . . . . . . . . . . . . . . . . . . . . . . . . . 219
9.2 Tipo enumerado . . . . . . . . . . . . . . . . . . . . . . . . . . . 221
9.2.1 Definición de tipos enumerados . . . . . . . . . . . . . . . 222
9.2.2 Uso de tipos enumerados . . . . . . . . . . . . . . . . . . . 222
9.3 El tipo predefinido `bool` . . . . . . . . . . . . . . . . . . . . . . 225
9.4 Tipos estructurados . . . . . . . . . . . . . . . . . . . . . . . . . 226
9.5 Tipo formación y su necesidad . . . . . . . . . . . . . . . . . . . 227
9.6 Tipo vector . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 228
9.6.1 Declaración de vectores . . . . . . . . . . . . . . . . . . . . 229
9.6.2 Inicialización de un vector . . . . . . . . . . . . . . . . . . 231
9.6.3 Operaciones con elementos de vectores . . . . . . . . . . . 231
9.6.4 Operaciones globales con vectores . . . . . . . . . . . . . . 232
9.6.5 Paso de argumentos de tipo vector . . . . . . . . . . . . . . 233
9.7 Vector de caracteres: Cadena (*string*) . . . . . . . . . . . . . . 235
9.8 Tipo tupla y su necesidad . . . . . . . . . . . . . . . . . . . . . . 238
9.9 Tipo registro (*struct*) . . . . . . . . . . . . . . . . . . . . . . . 239
9.9.1 Definición de registros . . . . . . . . . . . . . . . . . . . . . 240
9.9.2 Variables de tipo registro y su inicialización . . . . . . . . 241
9.9.3 Uso de registros . . . . . . . . . . . . . . . . . . . . . . . . 241
9.10 Ejemplos de programas . . . . . . . . . . . . . . . . . . . . . . . 243
9.10.1 Ejemplo: Cálculo del día de la semana de una fecha . . . 243
9.10.2 Frases palíndromas . . . . . . . . . . . . . . . . . . . . . . 248
9.10.3 Ejemplo: Cálculos con fracciones . . . . . . . . . . . . . . 250

**10 Ampliación de estructuras de control . . . . . . . . . . . . . . 255**
10.1 Estructuras complementarias de iteración . . . . . . . . . . . . . 255

10.1.1 Repetición: Sentencia DO . . . . . 256
10.1.2 Sentencia CONTINUE . . . . . 258
10.2 Estructuras complementarias de selección . . . . . 258
10.2.1 Sentencia SWITCH . . . . . 259
10.3 Equivalencia entre estructuras . . . . . 263
10.3.1 Selección por casos . . . . . 263
10.3.2 Bucle con contador . . . . . 264
10.3.3 Repetición . . . . . 265
10.4 Ejemplos de programas . . . . . 265
10.4.1 Ejemplo: Imprimir tickets de comedor . . . . . 265
10.4.2 Ejemplo: Gestión de tarjetas de embarque . . . . . 268
10.4.3 Ejemplo: Calculadora . . . . . 273

**Ejercicios sin resolver - II . . . . . 281**

**11 Estructuras de datos . . . . . 285**
11.1 Argumentos de tipo vector abierto . . . . . 285
11.1.1 Ejemplo: Contar letras y dígitos . . . . . 287
11.2 Formaciones anidadas. Matrices . . . . . 288
11.2.1 Declaración de matrices y uso de sus elementos . . . . . 289
11.2.2 Operaciones con matrices . . . . . 290
11.2.3 Ejemplo: Contrastar una imagen . . . . . 292
11.3 El esquema unión . . . . . 295
11.3.1 El tipo `union` . . . . . 296
11.3.2 Registros con variantes . . . . . 298
11.4 Esquemas de datos y esquemas de acciones . . . . . 299
11.5 Estructuras combinadas . . . . . 300
11.5.1 Formas de combinación . . . . . 301
11.5.2 Tablas . . . . . 303
11.5.3 Ejemplo: Gestión de tarjetas de embarque . . . . . 305

**12 Esquemas típicos de operación con formaciones . . . . . 315**
12.1 Esquema de recorrido . . . . . 316
12.1.1 Recorrido de matrices . . . . . 318
12.1.2 Recorrido no lineal . . . . . 319
12.2 Búsqueda secuencial . . . . . 321
12.3 Inserción . . . . . 322
12.4 Ordenación por inserción directa . . . . . 324
12.5 Búsqueda por dicotomía . . . . . 326
12.6 Simplificación de las condiciones de contorno . . . . . 328
12.6.1 Técnica del centinela . . . . . 328

12.6.2 Matrices orladas . . . . . . 330
12.7 Ejemplos de programas . . . . . . 333
12.7.1 Ejemplo: Sopa de letras . . . . . . 333
12.7.2 Ejemplo: Imprimir fechas en orden . . . . . . 338
12.7.3 Ejemplo: Recortar una imagen . . . . . . 345

**13 Punteros y variables dinámicas . . . . . . 351**
13.1 Estructuras de datos no acotadas . . . . . . 351
13.2 La estructura secuencia . . . . . . 352
13.3 Variables dinámicas . . . . . . 354
13.3.1 Punteros . . . . . . 354
13.3.2 Uso de variables dinámicas . . . . . . 356
13.4 Realización de secuencias mediante punteros . . . . . . 358
13.4.1 Operaciones con secuencias enlazadas . . . . . . 360
13.4.2 Ejemplo: Leer números y escribirlos en orden . . . . . . 363
13.5 Punteros y paso de argumentos . . . . . . 366
13.5.1 Paso de punteros como argumentos . . . . . . 366
13.5.2 Paso de argumentos mediante punteros . . . . . . 367
13.5.3 Ejemplo: Leer números y escribirlos en orden . . . . . . 370
13.6 Punteros y vectores en C y C++ . . . . . . 372
13.6.1 Nombres de vectores como punteros . . . . . . 373
13.6.2 Paso de vectores como punteros . . . . . . 374
13.6.3 Matrices y vectores de punteros . . . . . . 375

**14 Tipos abstractos de datos . . . . . . 377**
14.1 Concepto de tipo abstracto de datos (TAD) . . . . . . 377
14.2 Realización de tipos abstractos en **C±** . . . . . . 379
14.2.1 Definición de tipos abstractos como tipos registro (`struct`) 379
14.2.2 Ocultación . . . . . . 385
14.2.3 Ejemplo: Imprimir fechas en orden . . . . . . 386
14.3 Metodología basada en abstracciones . . . . . . 392
14.3.1 Desarrollo por refinamiento basado en abstracciones . . . 392
14.4 Ejemplo: Dibujar una *Curva-C* . . . . . . 395

**15 Módulos . . . . . . 401**
15.1 Concepto de módulo . . . . . . 401
15.1.1 Especificación y realización . . . . . . 402
15.1.2 Compilación separada . . . . . . 403
15.1.3 Descomposición modular . . . . . . 404
15.2 Módulos en **C±** . . . . . . 405
15.2.1 Proceso de compilación simple . . . . . . 406

15.2.2 Módulo principal . . . . . 407
15.2.3 Módulos no principales . . . . . 407
15.2.4 Uso de módulos . . . . . 410
15.2.5 Declaración y definición de elementos públicos . . . . . 411
15.2.6 Conflicto de nombres en el ámbito global . . . . . 412
15.2.7 Unidades de compilación en C± . . . . . 414
15.2.8 Compilación de programas modulares. Proyectos . . . . . 415
15.3 Desarrollo modular basado en abstracciones . . . . . 416
15.3.1 Implementación de abstracciones como módulos . . . . . 417
15.3.2 Dependencias entre ficheros. Directivas . . . . . 422
15.3.3 Datos encapsulados . . . . . 423
15.3.4 Reutilización de módulos . . . . . 429

**Ejercicios sin resolver - III . . . . . 433**

**A Sintaxis de C± . . . . . 435**
A.1 Unidad de compilación . . . . . 436
A.2 Directivas de programa . . . . . 436
A.3 Declaraciones globales . . . . . 436
A.4 Declaraciones de interfaz . . . . . 437
A.5 Constantes . . . . . 437
A.6 Tipos . . . . . 437
A.7 Variables . . . . . 438
A.8 Subprogramas . . . . . 438
A.9 Bloque de código . . . . . 439
A.10 Declaraciones de bloque . . . . . 439
A.11 Sentencias ejecutables . . . . . 439
A.12 Expresiones . . . . . 440
A.13 Elementos básicos . . . . . 441
A.14 Índice de reglas BNF . . . . . 442

**B Manual de Estilo . . . . . 445**
B.1 Aspectos generales . . . . . 446
B.1.1 Sintaxis . . . . . 446
B.1.2 Encolumnado . . . . . 446
B.1.3 Comentarios . . . . . 446
B.1.4 Identificadores . . . . . 447
B.2 Declaraciones . . . . . 447
B.2.1 Constantes . . . . . 448
B.2.2 Tipos de datos . . . . . 448
B.2.3 Variables . . . . . 450

B.2.4 Subprogramas . . . . . 450
B.2.5 Tipos abstractos de datos . . . . . 451
B.3 Sentencias . . . . . 452
B.4 Expresiones . . . . . 454
B.5 Punteros . . . . . 455
B.6 Módulos . . . . . 455

**C Notación lógico-matemática . . . . . 457**
C.1 Operadores numéricos . . . . . 457
C.2 Operadores de comparación . . . . . 458
C.3 Operadores lógicos . . . . . 458
C.4 Colecciones . . . . . 458
C.5 Cuantificadores . . . . . 459
C.6 Expresiones condicionales . . . . . 459

**Bibliografía . . . . . 461**

**Índice analítico . . . . . 463**

# Prólogo

## *Objetivos*

El objetivo fundamental de este libro es introducir de manera progresiva y sistemática una correcta metodología para la programación de computadores. Las materias que se cubren son las que se necesitan conocer en un curso de primer nivel de programación. Además, a lo largo del libro se van introduciendo de forma progresiva las estructuras y herramientas necesarias en cada momento y que están disponibles en cualquier lenguaje de programación de propósito general. No se ha considerado adecuado presentar un lenguaje en su totalidad dado que las estructuras no utilizadas quedan fuera del alcance de este libro.

El contenido ha sido pensado como libro de texto para una asignatura de Fundamentos de Programación del primer cuatrimestre del primer año de un Grado en Informática o similares dentro del marco de la UNED. Por tanto, se cuidan de manera especial los aspectos específicos de la enseñanza a distancia. Se trata de introducir los conceptos de manera progresiva, poco a poco, de manera que el alumno pueda ir avanzando a su ritmo. Cada concepto que se introduce se acompaña de las técnicas necesarias para su inmediata aplicación y ejemplos ilustrativos.

En este libro se utiliza como vehículo para la enseñanza de la programación el lenguaje **C±** (léase C-más-menos) que está constituido por un subconjunto del vocabulario de los lenguajes C y C++. Por lo tanto, cualquier programa escrito en el lenguaje **C±** se podrá editar, compilar y ejecutar en un entorno de desarrollo para C/C++. Se considera muy importante usar un lenguaje real para que el alumno acceda de manera natural e inmediata al computador. Esto permite al alumno comprobar en la práctica que los ejemplos propuestos funcionan.

Con la definición del lenguaje **C±** se ha buscado la creación de un lenguaje que facilite la enseñanza de la programación y que también se pueda utilizar

en el desarrollo de cualquier aplicación real. Las ventajas del lenguaje **C±** se pueden concretar en las siguientes:

- Es un lenguaje bien estructurado: que ha sido pensado para aplicar la metodología de programación estructurada, en sentido amplio. Los lenguajes C/C++ son excesivamente complejos para un primer curso de programación y en algunas ocasiones sus sentencias resultan complejas, ambiguas y poco claras. En el lenguaje **C±** no se incluyen todas las sentencias de C/C++ y además se imponen ciertas restricciones metodológicas en las sentencias utilizadas.
- El aprendizaje del lenguaje es relativamente sencillo dado que ha sido diseñado para la enseñanza de programación.
- **C±** soporta la programación modular y tipos abstractos de datos: ambos paradigmas de programación se consideran muy importantes para introducir al alumno en una buena metodología de diseño y desarrollo de programas de cierta complejidad.
- Las características antes mencionadas permiten que **C±** se pueda utilizar en cursos posteriores de programación. Por ejemplo, para presentar la programación orientada a objetos se debería incorporar a **C±** el concepto de clase y las estructuras de programación que ya están disponible en C++

En todo caso, el texto no se limita a enseñar un lenguaje, sino que trata que el alumno adquiera desde un principio una correcta metodología de programación, independiente del lenguaje utilizado. Así, se considera muy importante que el alumno adquiera una buena capacidad general de expresarse de manera formal, con independencia del lenguaje empleado. En todo momento se insiste en las técnicas de desarrollo por refinamiento progresivo.

Por otro lado, los desarrollos de grandes aplicaciones nunca los realiza un único programador. En este texto se consideran fundamental el empleo de buenas prácticas de ingeniería de software aplicadas a la programación. Cualquier empresa o equipo de desarrollo de software debe disponer antes del inicio de cada desarrollo de un “Manual de Estilo” para lograr la adecuada claridad, homogeneidad y mantenibilidad de los programas. El “Manual de Estilo” que se propone en este texto recopila un conjunto de buenas prácticas de programación que el alumno deberá seguir incluso para la realización de programas sencillos.

## *¿A quién va dirigido el texto?*

En principio, se trata de un texto pensado para una asignatura de Fundamentos de Programación del primer curso de un Grado en Informática o similares; por tanto va dirigido a los alumnos de primer año de estas carreras.

Sin embargo, podrá ser usado como texto de introducción a la programación por cualquier otra persona interesada en este tema. Los requisitos que se consideran necesarios para poder seguir adecuadamente el contenido del texto son los siguientes:

- Conocimientos generales de matemáticas, en especial de formalismos algebraicos.
- Capacidad para seguir un razonamiento lógico.
- Capacidad de organización.
- Aptitud para expresarse formalmente (dominio del lenguaje).

## *Metodología*

Para un correcto aprovechamiento del contenido del curso es imprescindible que el alumno tenga acceso a un computador con el compilador de **C±**. En el Manual de Prácticas asociado a este libro se dan las pautas para que el alumno pueda instalar y utilizar un entorno de programación configurado y adaptado especialmente para verificar la sintaxis de **C±** y que inmediatamente procede a compilar el programa con un compilador de C++. Además, el entorno de programación dispone de una herramienta para formatear el programa de acuerdo con las recomendaciones del "manual de estilo" de este libro.

Conviene que el alumno compruebe en el entorno de programación el correcto funcionamiento de algunos los ejemplos descritos en cada capítulo para someterlos a crítica. Esto le permitirá adquirir una capacidad de análisis previa a las tareas de diseño de sus propios programas. A continuación convendrá que el alumno realice los ejercicios propuestos también en el computador, y que compruebe igualmente su funcionamiento. En general no existe una solución única para un mismo problema. La solución debe ser examinada a posteriori para analizar además del correcto funcionamiento también los aspectos de claridad y estilo.

# Tema 1

# Introducción

El objetivo de este tema es introducir los conceptos generales, y dar una panorámica de la programación que permita posteriormente situar en el contexto adecuado las técnicas y metodologías que se expondrán en el resto del libro.

Especialmente importante es la presentación de diferentes modelos abstractos de cómputo, para poner de manifiesto que la programación imperativa, aunque sea la más extendida, no constituye la única manera de representar programas, y que no hay que identificar el concepto de programa con el de secuencia de órdenes.

## 1.1 Máquinas y programas

Intuitivamente podemos asociar el concepto de *máquina* a un dispositivo o instrumento físico capaz de realizar un cierto trabajo u operación. El concepto puede extenderse incluyendo máquinas que, aunque no existan físicamente, pueden concebirse y describirse con precisión y predecir su comportamiento. Estas máquinas se denominan *máquinas virtuales*.

### 1.1.1 Máquinas programables

En general, las máquinas operan a lo largo del tiempo, por lo que el concepto de máquina lleva asociado el de un proceso de funcionamiento en el cual diferentes operaciones se van realizando sucesiva o simultáneamente. Desde el punto de vista del control de su funcionamiento, podemos clasificar las máquinas en diferentes tipos.

Las *máquinas no automáticas*, o de control manual, son gobernadas por un operador o agente externo que desencadena unas determinadas operaciones en cada momento. Por ejemplo, una máquina de escribir imprime las letras o mueve el papel de acuerdo con las teclas pulsadas por el mecanógrafo.

Las *máquinas automáticas* actúan por sí solas, sin necesidad de operador, aunque pueden responder a estímulos externos. Por ejemplo, un ascensor automático gobierna por sí mismo los movimientos de subida y bajada incluyendo cambios de velocidad, apertura y cierre de puertas, etc., de forma coordinada, respondiendo a los estímulos de los botones de llamada o envío a un piso dado.

El funcionamiento de una máquina automática puede depender de la forma en que está construida, es decir, de los elementos que la componen y la manera en que están conectados entre sí. En este caso el comportamiento de la máquina será fijo, en el sentido de que a unos determinados estímulos externos responderá siempre de la misma manera. Esto ocurre en el ejemplo del ascensor.

Otras máquinas automáticas se denominan programables , y su comportamiento no es siempre el mismo. Una *máquina programable* (figura 1.1) se puede concebir como una máquina base, de comportamiento fijo, que se completa con una parte modificable que describe el funcionamiento de la máquina base. Esta parte modificable se denomina *programa*.

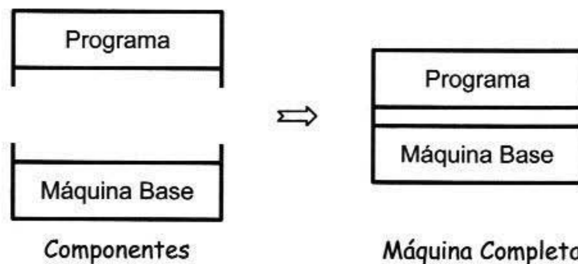



Figura 1.1 Componentes de una máquina programable.

Aunque habitualmente no se considere como tal, podemos analizar un reproductor de CD como una máquina programable, identificando el CD (reemplazable) con el programa. Incluso podemos establecer la siguiente serie de ejemplos:

- *Piano*: máquina manual de producir música.
- *Caja de música*: máquina automática de producir música (fija).
- *Reproductor de CD*: máquina programable de producir música (variable).

Dependiendo de cuál sea el programa que gobierne su funcionamiento, una máquina programable responderá a los estímulos externos de una forma o de otra. Una máquina programable se comporta, por tanto, como diferentes máquinas particulares, en función del programa utilizado.

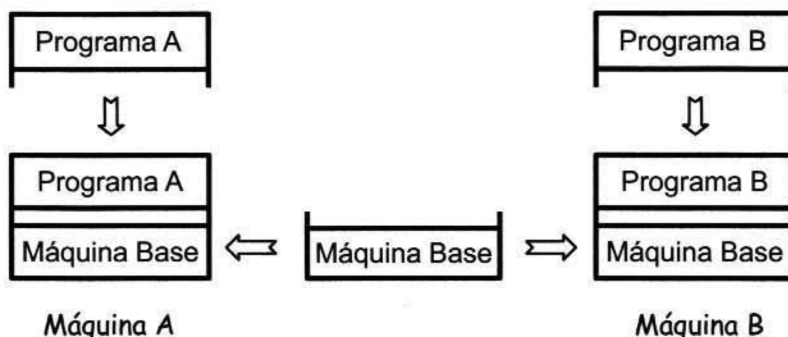



Figura 1.2 Una máquina programable puede comportarse como diferentes máquinas.

Cuando una máquina programable opera bajo control de un programa determinado, se dice que el programa se *ejecuta* en dicha máquina.

### 1.1.2 Concepto de cómputo

La palabra *cómputo* es sinónimo de cuenta o cálculo. Si consultamos un diccionario podemos encontrar una definición más elaborada:

**Cómputo** (del latín *computum* ). Determinación indirecta de una cantidad mediante el cálculo de ciertos datos.

En esta definición se puede apreciar que un cómputo es una operación de tratamiento de información. A partir de una información conocida se obtiene otra nueva como resultado de unos cálculos. En informática y de una forma general puede identificarse el concepto de cómputo con el de *tratamiento de la información.*

Un cómputo puede expresarse de diferentes maneras. Por ejemplo, mediante una fórmula o expresión matemática, tal como

$$34 \times 5 + 8 \times 7$$

Un cómputo se concibe también como un proceso a lo largo del cual se van realizando operaciones o cálculos elementales hasta conseguir el resultado final. En el ejemplo anterior se encuentra implícito dicho proceso. El resultado se podría obtener mediante los siguientes cálculos elementales:

1. Producto de 34 por 5, obteniendo 170.
2. Producto de 8 por 7, obteniendo 56.
3. Suma de los resultados anteriores, obteniendo 226.

En la expresión matemática usada como ejemplo están implícitos estos cálculos elementales, así como el orden en que pueden ser realizados. Los cálculos 1º y 2º podrían realizarse en cualquier orden, pero el cálculo 3º ha de realizarse necesariamente después de los otros dos.

### 1.1.3 Concepto de computador

La máquina programable por excelencia es el *computador*. Un computador se define como una máquina programable para tratamiento de la información, es decir, un computador es (¡obviamente!) una máquina para realizar cómputos.

Un programa de computador es, por tanto, una descripción de un cómputo. Al mismo tiempo nos encontramos con que un programa es también una descripción del comportamiento de una máquina, y podemos así considerarlo como una *máquina virtual* cuando convenga.

Un computador, como máquina programable que es, posee unos elementos fijos (máquina base) y otros modificables (programa). De forma simplificada podemos asociar los elementos fijos a los dispositivos físicos del computador, que constituyen el *hardware*, y los elementos modificables a las representaciones de los programas en sentido amplio, que constituyen el *software*.

Los computadores actuales corresponden a un tipo particular de máquinas programables que se denominan *máquinas de programa almacenado*. En estas máquinas la modificación del programa no implica un cambio de componentes físicos de la máquina, sino que estas máquinas poseen una memoria en la cual se puede almacenar información de cualquier tipo, debidamente codificada, y esta información incluye tanto los datos con los que opera la máquina como la representación codificada del programa. El programa es, por tanto, pura información, no algo material.

La estructura general de un computador se puede representar como se muestra en la figura 1.3. La memoria almacena datos y programas. Los dispositivos de entrada/salida permiten intercambiar información con el exterior, y el procesador es el elemento de control, que realiza operaciones elementales de tratamiento de la información interna, u operaciones de entrada o salida de información al exterior, de acuerdo con los códigos del programa que están almacenados en la memoria.

Figura 1.3 Esquema general de un computador.

## 1.2 Programación e ingeniería de software

De las explicaciones anteriores se deduce que una máquina programable, y en particular un computador, es totalmente inútil si no dispone del programa adecuado. Para realizar un determinado tratamiento de información con ayuda de un computador habrá sido necesario:

(a) Construir el computador (*hardware*).
(b) Idear y desarrollar el programa (*software*).
(c) Ejecutar dicho programa en el computador.

Sólo la última fase (c) es habitualmente realizada por el usuario. Las dos primeras corresponden a los profesionales de la informática: la fase (a) a los fabricantes de *hardware* y la (b) a los de *software*. En los siguientes apartados se analiza la actividad de desarrollo de software.

### 1.2.1 Programación

La labor de desarrollar programas se denomina en general *programación*. En realidad este término se suele reservar para designar las tareas de desarrollo de programas en pequeña escala, es decir, realizadas por una sola persona. El desarrollo de programas complejos, que son la mayoría de los usados actualmente, exige un equipo más o menos numeroso de personas que debe trabajar de manera organizada. Las técnicas para desarrollo de software a gran escala constituyen la *ingeniería de software*.

Programación e ingeniería de software no son disciplinas independientes, sino complementarias. El desarrollo de software en gran escala consiste esencialmente en descomponer el trabajo total de programación en partes independientes que pueden ser desarrolladas por miembros individuales del equipo. La ingeniería de software se limita a añadir técnicas o estrategias organizativas a las técnicas básicas de programación. El trabajo en equipo es, en último extremo, la suma de los trabajos realizados por los individuos.

### 1.2.2 Objetivos de la programación

La ingeniería de software excede del ámbito de este libro. En él nos centraremos sólo en la labor de programación, correspondiente a la preparación de programas medianos o pequeños, realizables por una sola persona. No obstante, las técnicas de programación han de establecerse con el objetivo de ser una base adecuada para la ingeniería de software. Entre los objetivos particulares de la programación podemos reconocer los siguientes:

- CORRECCIÓN: Es evidente que un programa debe realizar el tratamiento esperado, y no producir resultados erróneos. Esto tiene una consecuencia inmediata que no siempre se considera evidente: antes de desarrollar un programa debe especificarse con toda claridad cuál es el funcionamiento esperado. Sin dicha especificación es inútil hablar de funcionamiento correcto.
- CLARIDAD: Prácticamente todos los programas han de ser modificados después de haber sido desarrollados inicialmente. Por esta razón es fundamental que sus descripciones sean claras y fácilmente inteligibles por otras personas distintas de su autor, o incluso por el mismo autor al cabo de un cierto tiempo, cuando ya ha olvidado los detalles del programa.
- EFICIENCIA: Una tarea de tratamiento de información puede ser programada de muy diferentes maneras sobre un computador determinado, es decir, habrá muchos programas distintos que producirán los resultados deseados. Algunos de estos programas serán más eficientes que otros. Los programas eficientes aprovecharán mejor los recursos disponibles y, por tanto, su empleo será más económico en algún sentido.

Estos y otros objetivos resultan a veces contrapuestos. Quizá el ejemplo más intuitivo sea la dualidad entre claridad y eficiencia. Para ser claros los programas han de ser sencillos, pero para aprovechar los recursos de manera eficiente en muchos casos hay que introducir complicaciones que hacen el programa más difícil de entender.

Si se trata de establecer una importancia relativa entre los distintos objetivos, habría que considerar como prioritaria la *corrección*. Piénsese, por ejemplo, que un programa de contabilidad no es aceptable si no calcula correctamente los saldos de las cuentas.

A continuación debe perseguirse la *claridad*, que como ya se ha indicado es necesaria para poder realizar modificaciones, o simplemente para poder certificar que el programa es correcto. En realidad el objetivo de claridad va ligado al de corrección. Es prácticamente imposible asegurar que un programa es correcto si no puede ser entendido claramente por la persona que lo examina.

Tal como se ha dicho antes, la claridad facilita la tarea de realizar modificaciones cuando las necesidades así lo exijan. Puede afirmarse que esto ocurre siempre con todos los programas que tienen un cierto interés.

Finalmente ha de atenderse a la *eficiencia*. Este objetivo, aunque importante, sólo suele ser decisivo en determinados casos. En muchas situaciones el aumento de capacidad de los computadores a medida que avanza la tecnología va permitiendo utilizar de manera aceptable, desde el punto de vista económico, programas relativamente menos eficientes.

## 1.3 Lenguajes de programación

Ya se ha explicado que un computador funciona bajo control de un programa que ha de estar almacenado en la unidad de memoria. El programa contiene una descripción codificada del comportamiento deseado del computador.

Cada modelo de computador podrá utilizar una forma particular de codificación de programas, que no coincidirá con la de otros modelos. La forma de codificar programas de una máquina en particular se dice que es su *código de máquina* o *lenguaje de máquina*. La palabra "lenguaje" utilizada habitualmente en el vocabulario informático en español es, en realidad, una transcripción directa del término inglés "*language*", cuyo significado correcto es "idioma".

Un programa codificado en el lenguaje de un modelo de máquina (figura 1.4) no podrá ser ejecutado, en general, en otro distinto. Si queremos que un programa funcione en diferentes máquinas tendremos que preparar versiones particulares en el lenguaje de máquina de cada una de ellas. Evidentemente con ello se multiplica el costo de desarrollo si cada versión se prepara de manera independiente.

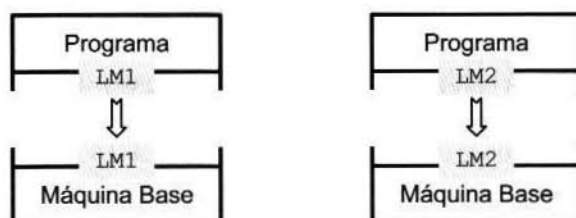



Figura 1.4 Cada computador necesita programas en su lenguaje de máquina (LM) particular.

Por otra parte, los programas en código de máquina son extraordinariamente difíciles de leer por una persona. Normalmente contienen códigos numéricos

(figura 1.5) sin ningún sentido nemotécnico, y compuestos por millones de operaciones elementales muy sencillas que en conjunto pueden realizar los tratamientos complejos que vemos a diario.

```
88 94 50 FF 76 0A FF 76 08 9A BA CD 3A 16 B8 01
00 EB E8 B8 88 94 50 2B C0 50 9A FA C5 3A 16 EB
ED B8 88 94 50 B8 01 00 EB EF B8 88 94 50 9A 48
D1 3A 16 EB D9 5D CA 0A 00 55 8B EC 83 EC 08 57
56 B8 01 00 50 9A 97 41 9B 34 8B D8 8B 47 14 89
```

Figura 1.5 Fragmento de programa en código de máquina.

Para facilitar la tarea de programación resulta muy deseable disponer de formas de representación de los programas que sean adecuadas para ser leídas o escritas por personas. En particular los *lenguajes de programación* sirven precisamente para representar programas de manera simbólica, en forma de un texto (figura 1.6) que puede ser leído con relativa facilidad por una persona. Además los lenguajes de programación son formas de representación prácticamente independientes de las máquinas particulares que se vayan a usar.

```
void PintarPlazas(const TipoPlazas P) {

  printf("\n\n");
  printf("     A   B   C      D   E    F\n\n");
  for (int i = 0; i < NumFilas; i++) {
    printf("%3d",i+1);
    for (int j = 0; j < AsientosFila; j++) {
      if ( j == Pasillo ) {
        printf("   ");
      }
      if (P[i].AsientosOcupa[j] == ocupado) {
        printf("  (*)");
      } else if (P[i].AsientosOcupa[j] == reservado) {
        printf("  (R)");
      } else if (P[i].AsientosOcupa[j] == vacio) {
        printf("  ( )");
      }
    }
    printf("\n");
  }
  printf("\n");
}
```

Figura 1.6 Fragmento de programa en lenguaje **C±**.

La comparación de los fragmentos de programa de las Figuras 1.5 y 1.6 pone de manifiesto sin necesidad de más explicaciones la ventaja de usar lenguajes de programación simbólicos.

## 1.4 Compiladores e Intérpretes

Un programa escrito en un lenguaje de programación simbólico puede ser ejecutado en máquinas muy diferentes. Pero para ello se necesita disponer de los mecanismos adecuados para transformar ese programa simbólico (figura 1.7) en un programa en el lenguaje particular de cada máquina.

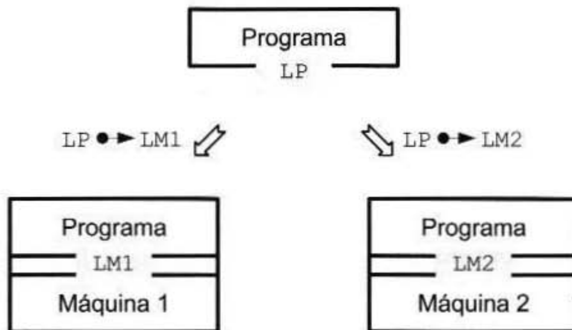



Figura 1.7 Un programa en un lenguaje de programación simbólico ha de adaptarse al lenguaje de cada máquina.

Existen diferentes estrategias para conseguir ejecutar en una máquina determinada un programa escrito en un lenguaje de programación simbólico. Normalmente se basan en el uso de programas especiales que realizan un tratamiento de la información en forma de texto que representa el programa en el lenguaje de programación simbólico. Estos programas para manipular representaciones de programas los denominaremos *procesadores de lenguajes.*

Un *compilador* es un programa que traduce programas de un lenguaje de programación simbólico a código de máquina. A la representación del programa en lenguaje simbólico se le llama *programa fuente*, y su representación en código de máquina se le llama *programa objeto.* Análogamente al lenguaje simbólico y al lenguaje máquina se les llama también *lenguaje fuente* y *lenguaje objeto*, respectivamente.

La ejecución del programa mediante compilador exige al menos dos etapas separadas, tal como se indica en la figura 1.8. En la primera de ellas se traduce el programa simbólico a código de máquina mediante el programa compilador.

En la segunda etapa se ejecuta ya directamente el programa en código de máquina, y se procesan los datos y resultados particulares. La compilación del programa ha de hacerse sólo una vez, quedando el programa en código de máquina disponible para ser utilizado de forma inmediata tantas veces como se desee.

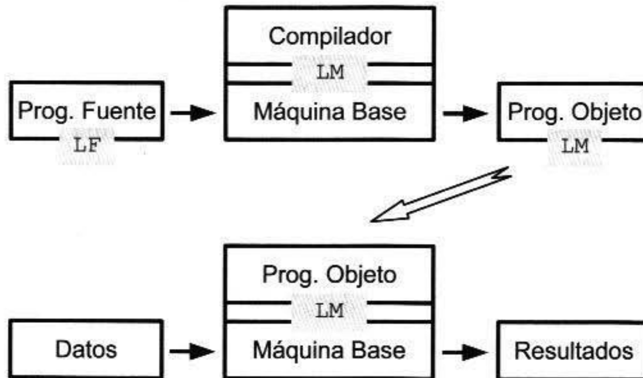



Figura 1.8 Proceso de un programa en lenguaje fuente (LF) mediante compilador.

Un *intérprete* es un programa que analiza directamente la descripción simbólica del programa fuente y realiza las operaciones oportunas. El intérprete debe contener dentro de él los fragmentos de código de máquina de todas las operaciones posibles que se puedan usar en el lenguaje de programación simbólico. Puede decirse que el intérprete es (o simula) una *máquina virtual* (figura 1.9) cuyo lenguaje de máquina coincide con el lenguaje fuente.

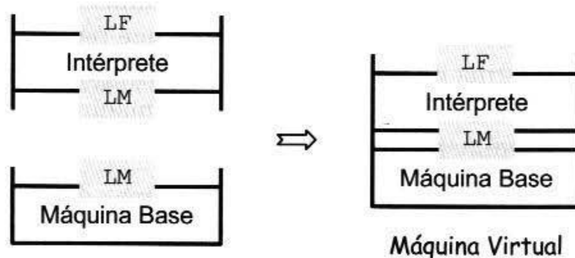



Figura 1.9 Un intérprete se comporta como una máquina virtual cuyo lenguaje es el lenguaje fuente.

El proceso de un programa mediante intérprete (figura 1.10) se limita a ejecutar directamente el programa en la máquina virtual, es decir, sobre el intérprete.

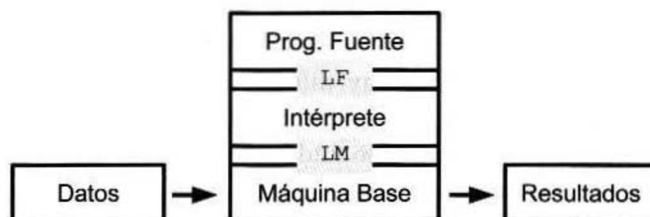



Figura 1.10 Proceso de un programa en lenguaje fuente (LF) mediante intérprete.

El proceso mediante intérprete es más sencillo, en conjunto, que mediante compilador, ya que no hay que realizar dos fases separadas. Su principal inconveniente es que la velocidad de ejecución es más lenta, ya que al tiempo que se van tratando los datos de la aplicación hay que ir haciendo el análisis e interpretación de las operaciones descritas en el programa fuente.

## 1.5 Modelos abstractos de cómputo

Los lenguajes de programación permiten describir programas o cómputos de manera formal, y por tanto simbólica y rigurosa. La descripción se hace, naturalmente, basándose en determinados elementos básicos y formas de combinación de estos elementos simples para construir programas tan complicados como sea necesario.

Existen muchísimos lenguajes de programación distintos que unas veces difieren en aspectos generales y otras simplemente en detalles. Si analizamos estos lenguajes podremos observar que muchos de ellos utilizan elementos básicos y formas de combinación similares, aunque representándolos con símbolos diferentes.

Si de un conjunto de lenguajes de programación basados en elementos computacionales similares extraemos los conceptos comunes, obtendremos un *modelo abstracto de cómputo.* Este modelo abstracto recoge los elementos básicos y formas de combinación de una manera abstracta, prescindiendo de la notación concreta usada en cada lenguaje de programación para representarlos.

Existen diversos modelos abstractos de cómputo, o modelos de programación, que subyacen en los lenguajes de programación actuales. Entre ellos están la *programación funcional*, *programación lógica*, *programación imperativa*, *modelo de flujo de datos*, *programación orientada a objetos*, etc. Todos estos modelos son modelos universales, en el sentido de que pueden utilizarse para describir cualquier cómputo intuitivamente posible.

Quizá el aspecto más interesante a destacar en este análisis de modelos abstractos de cómputo es que un programa que permita resolver un determinado problema puede adoptar formas muy diferentes, dependiendo del modelo de cómputo que se utilice para desarrollarlo. El modelo de programación imperativa es el más extendido, y eso induce a quienes empiezan a estudiar informática a identificar el concepto de programa con el de secuencia o lista de órdenes. Sin embargo, como veremos a continuación, existen otras formas igualmente válidas de representar un programa.

### 1.5.1 Modelo funcional

El modelo de *programación funcional* se basa casi exclusivamente en el empleo de funciones. El concepto de función se corresponde aquí de manera bastante precisa con el concepto de función en matemáticas. Una función es una aplicación, que hace corresponder un elemento de un conjunto de destino (resultado) a cada elemento de un conjunto de partida (argumento) para el que la función esté definida.

Por ejemplo, la operación de suma de números enteros es una función en que el conjunto de partida es el de las parejas de números enteros y el de destino es el conjunto de los números enteros. A cada pareja de enteros se le hace corresponder un entero, que es su suma.

De forma convencional, representaremos como $f(x)$ al resultado que se obtendrá al aplicar la función $f$ al argumento $x$. Por ejemplo, podemos suponer definidas las funciones de suma, resta y producto de la forma:

| **Función** | **Resultado** |
|---|---|
| Suma( a, b ) | a + b |
| Diferencia( a, b ) | a - b |
| Producto( a, b ) | a × b |

Para describir cómputos complejos, las funciones pueden combinarse unas con otras, de manera que el resultado obtenido en una función se use como argumento para otra. De esta manera un cómputo tal como

$$34 \times 5 + 8 \times 7$$

puede representarse de manera funcional de la forma

Suma( Producto( 34, 5 ), Producto( 8, 7 ) )

Este es el aspecto que tiene un programa funcional, que será siempre en último extremo una aplicación de una función a unos argumentos, para obtener un resultado. El proceso de cómputo, llamado *reducción*, se basa en reemplazar

progresivamente cada función por el resultado de la misma. Este sistema de evaluación por sustitución es la base del llamado *cálculo-λ*. Aplicado al ejemplo se tendría:

| Cómputo parcial | Expresión / Resultado |
|---|---|
| | Suma( Producto( 34, 5), Producto( 8, 7) ) |
| $34 \times 5$ | Suma( 170, Producto( 8, 7) ) |
| $8 \times 7$ | Suma( 170, 56 ) |
| $170 + 56$ | 226 |

Las explicaciones anteriores se refieren a cómputos en los que sólo intervienen funciones primitivas, que son las que el computador o máquina abstracta que ejecuta el programa puede evaluar de forma directa. La programación funcional permite la definición por parte del programador de nuevas funciones a partir de las ya existentes. Utilizando de manera convencional el símbolo ::= para indicar definición, podremos crear una nueva función, *Cuadrado*, para obtener el cuadrado de un número basándose en el uso del producto de dos números.

Cuadrado( x ) ::= Producto( x, x )

Cuando en un cómputo intervienen funciones definidas, la evaluación se sigue haciendo por sustitución. El proceso, llamado *reescritura*, consiste en reemplazar una función por su definición, sustituyendo los argumentos simbólicos en la definición por los argumentos reales en el cómputo. Por ejemplo, para evaluar $(5+3)^2$ tendremos:

| Cómputo parcial | Expresión / Resultado |
|---|---|
| | Cuadrado( Suma( 5, 3 ) ) |
| reducir Suma | Cuadrado( 8 ) |
| reescribir Cuadrado | Producto( 8, 8 ) |
| reducir Producto | 64 |

### 1.5.2 Modelo de flujo de datos

En este modelo de cómputo, un programa corresponde a una *red de operadores* interconectados entre sí. Cada operador lo representaremos gráficamente mediante un cuadrado con *entradas* y *salidas*, y dentro de él el símbolo de la operación que realiza. Un operador espera hasta tener valores presentes en sus entradas, y entonces se activa él solo, consume los valores en las entradas, calcula el resultado, y lo envía a la salida. Después de esto vuelve a esperar que le lleguen nuevos valores por las entradas.

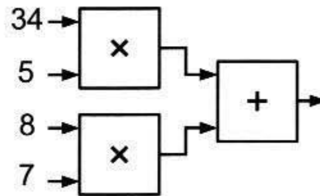



Figura 1.11 Red de flujo de datos.

Por ejemplo, la expresión $34 \times 5 + 8 \times 7$ puede ser calculada por la red de la figura 1.11.

El cómputo se realiza durante la evolución de la red, tal como se indica en la figura 1.12. Cuando la evolución termina, el resultado está presente en la salida de la derecha.

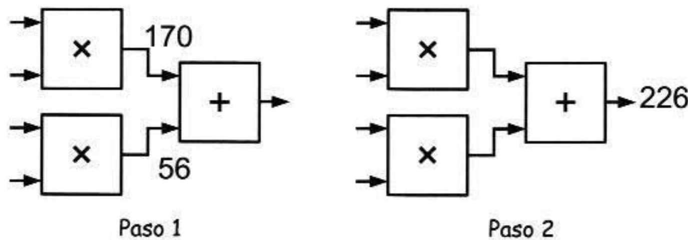



Figura 1.12 Evolución de la red de flujo de datos.

Una red de flujo de datos puede organizarse de manera que opere de forma iterativa, obteniendo no ya un resultado sino una serie de ellos. Por ejemplo, la red de la figura 1.13 produce la serie de números naturales, a base de reciclar sobre sí mismo un operador de incremento. Las líneas verticales representan operadores de duplicación o mezcla de valores.

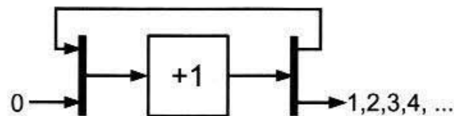



Figura 1.13 Generación de una serie de números.

Esta red no termina nunca de evolucionar. Añadiendo operadores especiales de bifurcación se puede conseguir que se detenga al llegar a un resultado adecuado.

### 1.5.3 Modelo de programación lógica

Este modelo abstracto de cómputo corresponde plenamente a lo que se denomina *programación declarativa*. Un programa consiste en plantear de manera formal un problema a base de declarar una serie de elementos conocidos, y luego preguntar por un resultado, dejando que sea la propia máquina la que decida cómo obtenerlo.

En *programación lógica* los elementos conocidos que pueden declararse son hechos y reglas. Un *hecho* es una relación entre objetos concretos. Una *regla* es una relación general entre objetos que cumplen ciertas propiedades. Una relación entre objetos la escribiremos poniendo el nombre de dicha relación y luego los objetos relacionados entre paréntesis. Por ejemplo:

Hijo( Juan, Luis )

significaría que Juan es hijo de Luis. Si tenemos el árbol genealógico como el de la figura 1.14

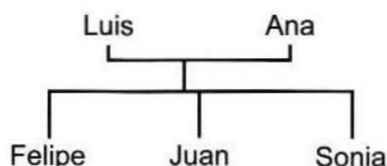



Figura 1.14 Árbol genealógico.

podremos declarar (prescindiendo del sexo) los hechos siguientes:

| Hechos |
|---|
| Hijo( Felipe, Luis ) |
| Hijo( Juan, Luis ) |
| Hijo( Sonia, Luis ) |
| Hijo( Felipe, Ana ) |
| Hijo( Juan, Ana ) |
| Hijo( Sonia, Ana ) |

Para realizar una consulta escribiremos el esquema de un hecho en que alguno de los elementos sea desconocido. Esto lo indicaremos usando nombres de incógnitas en minúscula, para distinguirlos de los nombres de elementos conocidos, que en este caso se habían escrito en mayúsculas (en lenguaje Prolog se usa el convenio contrario). La consulta será respondida indicando todos los valores posibles que puedan tomar las incógnitas. Por ejemplo:

| Consulta | Respuesta |
|---|---|
| Hijo( x, Ana ) | x = Felipe |
| | x = Juan |
| | x = Sonia |

La verdadera potencia de la programación lógica aparece cuando declaramos reglas. Al realizar consultas basadas en reglas la máquina realiza automáticamente las inferencias (deducciones) necesarias para responderla. Por ejemplo, usando el símbolo ":−" para definir una regla, escribiremos:

| Reglas |
|---|
| Padre( x, y ) :− Hijo( y, x ) |
| Hermano( x, y ) :− Hijo( x, z ), Hijo( y, z ) |

La primera regla se limita a nombrar la relación inversa de "hijo". La segunda expresa que dos personas son hermanas si son hijas de un mismo padre o madre. Ahora pueden realizarse consultas como las siguientes:

| Consulta | Respuesta |
|---|---|
| Padre( x, Sonia ) | x = Luis |
| | x = Ana |
| Hermano( x, Felipe ) | x = Felipe |
| | x = Juan |
| | x = Sonia |

La segunda consulta tiene una respuesta aparentemente errónea. Un análisis detallado nos permite comprobar que el error está en la formulación de la regla, ya que no se le ha informado a la máquina de que una persona no se considera hermana de sí misma. Modificando la regla se obtendrá la respuesta deseada.

| Reglas |
|---|
| Hermano( x, y ) :− Hijo( x, z ), Hijo( y, z ), ≠( x, y ) |

| Consulta | Respuesta |
|---|---|
| Hermano( x, Felipe ) | x = Juan |
| | x = Sonia |

### 1.5.4 Modelo imperativo

El modelo de programación imperativa responde a la estructura interna habitual de un computador, que se denomina *arquitectura Von Neumann*. Un

programa en lenguaje de máquina aparece como una lista de *instrucciones* u órdenes elementales que han de ejecutarse una tras otra, en el orden en que aparecen en el programa. El nombre de *programación imperativa* deriva del hecho de que un programa aparece como una lista de órdenes a cumplir.

El orden de ejecución puede alterarse en caso necesario mediante el uso de instrucciones de control. Con ello se consigue ejecutar o no, o repetir, determinadas partes del programa dependiendo de ciertas condiciones en los datos.

Las instrucciones de un programa imperativo utilizan datos almacenados en la memoria del computador. Esta capacidad de almacenamiento de valores se representa en los programas imperativos mediante el uso de *variables*. Una variable no tiene aquí el mismo significado que en matemáticas, sino que representa un dato almacenado bajo un nombre dado. Una variable contiene un valor que puede ser usado o modificado tantas veces como se desee.

Un programa imperativo se plantea como el cálculo o modificación de sucesivos valores intermedios hasta obtener el resultado final. Las instrucciones típicas de un programa imperativo son las de asignación, que consisten en obtener un resultado parcial mediante un cálculo elemental que puede ser realizado por la máquina, y que se almacena en una variable para ser utilizado posteriormente.

En los lenguajes de programación simbólicos las instrucciones u órdenes se denominan *sentencias*. En **C±** y otros lenguajes similares la sentencia de asignación se representa de la forma

$$variable = expresión$$

La parte derecha de esta sentencia es una expresión aritmética que puede usar variables o valores constantes, así como operadores que estén definidos en el lenguaje, tales como los correspondientes a las operaciones aritméticas habituales: suma, resta, etc. Una sentencia de asignación representa una orden de calcular el resultado de la expresión y luego almacenar dicho resultado como nuevo valor de la variable.

Usando sólo expresiones simples y variables auxiliares, podremos expresar el cálculo de

$$34 \times 5 + 8 \times 7$$

mediante las sentencias siguientes

**1.** $\mathrm{a} = 34 \times 5$
**2.** $\mathrm{b} = 8 \times 7$
**3.** $\mathrm{c} = \mathrm{a} + \mathrm{b}$

que obtendrán el resultado final en la variable $c$.

En realidad los lenguajes de programación permiten escribir directamente expresiones complejas. El cálculo anterior podría haberse hecho con una sola sentencia

**1.** c = 34 × 5 + 8 × 7

Para mostrar cómo las variables en programación imperativa pueden ir modificando su valor paso a paso hasta obtener el resultado deseado, analizaremos el siguiente fragmento de programa, que calcula el valor de $5! = 1\times2\times3\times4\times5$. En este programa se ha supuesto que existen instrucciones REPETIR y HASTA que permiten programar la repetición controlada de una parte del programa.

**1.** f = 1
**2.** k = 1
**3.** REPETIR
**4.** k = k + 1
**5.** f = f × k
**6.** HASTA k == 5

Este programa obtiene el resultado final en la variable $f$. La variable $k$ se utiliza para ir disponiendo de los sucesivos valores 2, 3, 4, etc. Las instrucciones **3.** y **6.** controlan la repetición de **4.** y **5.** La instrucción:

**4.** k = k + 1

tiene el significado siguiente: Se calcula el resultado de la expresión $k + 1$, usando el valor de $k$ al iniciarse la ejecución de esa instrucción, y el valor obtenido se almacena de nuevo en $k$ reemplazando el valor anterior. La primera vez que se ejecuta esa instrucción $k$ tiene el valor 1, asignado inicialmente por la instrucción **2.** Al sumarle 1 se obtiene el valor 2, y la variable $k$ pasa a tener ahora este nuevo valor. Cada vez que se vuelva a ejecutar la instrucción **4.** la variable $k$ incrementará su valor en 1 unidad.

El análisis detallado del funcionamiento de un programa imperativo puede hacerse mediante una traza en la que se van anotando las sentencias o instrucciones de asignación que se van ejecutando sucesivamente, y los valores que toman las variables inicialmente y tras cada instrucción. En este ejemplo se tendrá:

| Instrucción | k | f |
|---|---|---|
| | ? | ? |
| **1.** | ? | 1 |
| **2.** | 1 | 1 |
| **4.** | 2 | 1 |
| **5.** | 2 | 2 |
| **4.** | 3 | 2 |
| **5.** | 3 | 6 |
| **4.** | 4 | 6 |
| **5.** | 4 | 24 |
| **4.** | 5 | 24 |
| **5.** | 5 | 120 |

El programa comienza con valores no definidos (?) para las variables. Termina cuando $k$ ha tomado el valor 5, en cuyo caso finalizan las repeticiones y $f$ tiene el valor $120 = 5!$.

## 1.6 Elementos de la programación imperativa

La mayoría de los lenguajes de programación actualmente en uso siguen el modelo de programación imperativa. Por esta razón se ha optado en este primer nivel de enseñanza de programación por seguir dicho modelo. El lenguaje de programación **C±** se utilizará en este libro como herramienta para desarrollar las ideas generales en ejemplos prácticos realizables en máquina. **C±** es un subconjunto de los lenguajes C y C++ que se utilizan habitualmente en desarrollos reales. Además, **C±** presenta importantes ventajas para la enseñanza, por ser un lenguaje bien estructurado, permitir el uso de programación modular, permitir la implementación de tipos abstractos de datos, y ser un primer paso hacia el uso de lenguajes más evolucionados (y más complicados) como ocurre con los lenguajes C++, Ada o Java.

En el resto de este texto se irán desarrollando las ideas abstractas o generales de programación, junto con su realización en lenguaje **C±**. Todo ello en el marco de la programación imperativa, cuyos elementos abstractos se describen a continuación.

### 1.6.1 Procesador, entorno, acciones

Al introducir el modelo de programación imperativa se ha definido un programa, de manera intuitiva, como una lista de órdenes o instrucciones que han de ir siendo ejecutadas por la máquina en el orden preciso que se indique.

La idea abstracta correspondiente al concepto físico de máquina es el *procesador*. Definiremos como procesador a todo agente capaz de entender las órdenes del programa y ejecutarlas.

El procesador es esencialmente un elemento de control. Para ejecutar las instrucciones empleará los recursos necesarios, que formarán parte del sistema en el cual se ejecute el programa. Por ejemplo, se necesitarán dispositivos de almacenamiento para guardar datos que habrán de ser utilizados posteriormente, o dispositivos de entrada-salida que permitirán tomar datos de partida del exterior y presentar los resultados del programa. Todos estos elementos disponibles para ser utilizados por el procesador constituyen su *entorno*.

Las órdenes o instrucciones del programa definen determinadas *acciones* que deben ser realizadas por el procesador. Un programa imperativo aparece así como la descripción de una serie de acciones a realizar en un orden preciso. Las acciones son la idea abstracta equivalente a las instrucciones de un programa real.

### 1.6.2 Acciones primitivas. Acciones compuestas

Las acciones que son directamente realizables por el procesador se denominan *acciones primitivas*. Estas acciones suelen ser bastante sencillas, incluso en el caso de programas descritos en lenguajes de programación simbólicos. Entender un programa descrito enteramente a base de acciones primitivas suele ser muy difícil, por el nivel de detalle con el que hay que analizar cada una de sus partes, y todas ellas en conjunto.

El planteamiento razonable de un programa complejo debe pasar por usar la idea de *abstracción* para limitar la complejidad de detalles al estudiar el programa en su conjunto. Es muy útil usar la idea de *acción compuesta* como abstracción equivalente a un fragmento de programa más o menos largo que realiza una operación bien definida.

La descripción de un programa en términos de acciones compuestas puede facilitar su comprensión. Por supuesto, al desarrollar el programa habrá sido preciso describir o descomponer las acciones compuestas en otras más sencillas, hasta llegar finalmente a acciones primitivas, que son las que realmente podrá ejecutar el procesador. Las acciones compuestas son un elemento de descripción que facilita la comprensión del programa en su conjunto, o de partes importantes de él, pero no reduce el tamaño total del programa, que necesariamente deberá ser largo si se han de realizar operaciones complicadas.

### 1.6.3 Esquemas de acciones

Una acción compuesta consistirá, tal como se ha indicado, en la ejecución combinada de otras acciones más sencillas. La manera en la que varias acciones sencillas se combinan para realizar una acción complicada se denomina *esquema* de la acción compuesta.

Una buena metodología de programación exige usar esquemas sencillos y fáciles de entender a la hora de desarrollar acciones compuestas. A lo largo de este libro se irán introduciendo los principales esquemas de programación imperativa, junto con recomendaciones para su aplicación en el desarrollo de programas.

En particular, la llamada *programación estructurada* sugiere el uso de tres esquemas generales denominados *secuencia*, *selección* e *iteración*, con los cuales (junto con la definición de operaciones abstractas) se puede llegar a desarrollar de forma comprensible un programa tan complicado como sea necesario.

## 1.7 Evolución de la programación

Las ideas sobre cuál es la manera apropiada de desarrollar programas han ido evolucionando con el tiempo. Ha habido diversos motivos para ello, que pasaremos a analizar brevemente.

### 1.7.1 Evolución comparativa Hardware/Software

Los primeros computadores eran máquinas extraordinariamente costosas, y con una capacidad que hoy día consideraríamos ridículamente limitada. Sin embargo en su momento representaban el límite en la capacidad de tratamiento de información, y hacían posibles determinados trabajos de cálculo inabordables hasta entonces.

Como consecuencia de ello la finalidad principal de la programación era obtener el máximo rendimiento de los computadores. Los programas se escribían directamente en el lenguaje de la máquina, o a lo sumo en un lenguaje *ensamblador* en que cada instrucción de máquina se representaba simbólicamente mediante un código nemotécnico para facilitar la lectura del programa.

No existían ideas abstractas sobre el significado u objetivo preciso de un programa. Simplemente se consideraba que el mejor programa era el que realizaba el trabajo en menos tiempo y usando el mínimo de recursos de la máquina. La programación era, por tanto, una labor artesana, basada en la habilidad

personal del programador para conseguir que el programa cumpliera con esos objetivos de eficiencia, para lo cual era imprescindible conocer en detalle el funcionamiento interno del computador, y poder así emplear ciertos "trucos" de codificación que permitían ahorrar algunas instrucciones o usar menos elementos de memoria para almacenar datos intermedios.

El costo de desarrollo del software resultaba, en todo caso, muy inferior al costo del equipo material (*hardware*), por lo que pocas personas se paraban a considerar las posibilidades de reducir los costos de desarrollo de programas.

Por otra parte, y dada la limitación de capacidad de las máquinas, los programas eran necesariamente sencillos, en términos relativos, y se consideraba que podían llegar a depurarse de errores mediante ensayos o pruebas en número suficiente.

Los avances en la tecnología electrónica han ido suministrando computadores cada vez más capaces y baratos, en términos relativos. La necesidad de preparar sistemáticamente programas muy eficientes ha ido disminuyendo, y poco a poco se ha ido haciendo rentable utilizar programas no tan eficientes.

La mayor capacidad de los computadores ha permitido abordar aplicaciones cada vez más complejas. En los primeros computadores el software de una aplicación podía contener algunos cientos o quizá miles de instrucciones. En la actualidad las aplicaciones consideradas sencillas tienen decenas de miles de instrucciones, y en aplicaciones de gran envergadura el volumen del software desarrollado se cuenta por millones de instrucciones y en ellos trabajan centenares de programadores.

Con estos volúmenes de programa el costo del desarrollo del software supera ampliamente al costo de los equipos hardware utilizados. Ya no tiene sentido dedicar un gran esfuerzo a conseguir programas eficientes. Es más barato desarrollar programas relativamente más simples, aunque no aprovechen muy bien los recursos de la máquina, y comprar un computador de mayor potencia de proceso que compense esa posible falta de eficiencia. Los lenguajes de programación simbólicos han facilitado extraordinariamente las tareas de programación, al poder invocar en un programa operaciones cada vez más complejas como acciones primitivas. De esta manera se reduce el volumen total del programa medido en número de instrucciones, que ahora pasan a ser sentencias del lenguaje simbólico.

### 1.7.2 Necesidad de metodología y buenas prácticas

Los programas actuales son tan complicados que ya no es posible desarrollarlos de una manera artesanal. Es necesario aplicar técnicas de desarrollo muy

precisas para controlar el producto obtenido. Ya se ha indicado que estas técnicas aplicables a proyectos desarrollados en equipo constituyen la *ingeniería de software*.

A nivel individual hay que promover el empleo de una *metodología de programación* apropiada, que satisfaga los objetivos de corrección y claridad mencionados anteriormente. Para aplicaciones grandes la claridad se convierte en un objetivo prioritario, ya que resulta imposible analizar y modificar un programa si no se comprende suficientemente su funcionamiento.

Para facilitar la obtención de programas correctos, sin fallos, se pueden emplear técnicas formales, que permitan en lo posible garantizar la corrección del programa mediante demostraciones lógico-matemáticas, y no mediante ensayos en busca de posibles errores. La técnica de ensayos sólo resulta útil si consigue descubrir fallos, pues así demuestra que el programa contiene errores que hay que corregir, pero es más bien inútil si no se descubre ningún fallo, porque eso no garantiza que el programa no contenga errores, los cuales pueden manifestarse (y lamentablemente se manifiestan con harta frecuencia) cuando el programa ya está en explotación y los usuarios lo emplean en multitud de situaciones nuevas que no habían sido ensayadas nunca.

Lamentablemente las técnicas formales son tan complejas que sólo son utilizables en programas críticos de pequeño tamaño. Para grandes aplicaciones en las que intervienen cientos de ingenieros es absolutamente necesario emplear técnicas de ingeniería de software basadas en las *buenas prácticas*. Las buenas prácticas son un conjunto de normas, basadas en la experiencia de desarrollos anteriores, que se autoimponen todos los miembros de un equipo. El *Manual de Estilo* es el documento que compendia el conjunto de buenas prácticas que todos los miembros del equipo deben utilizar de una manera disciplinada durante el desarrollo de una aplicación compleja.

# Tema 2

# Elementos básicos de programación

En este tema se presenta un conjunto mínimo de elementos de un lenguaje de programación imperativo. Este conjunto se particulariza para el lenguaje **C±**. Con los elementos presentados se podrán construir programas completos aunque con una estructura muy simple, ya que sólo pueden estar formados por una secuencia de sentencias. Para que estos primeros programas produzcan resultados, se introducen también varios mecanismos de escritura simple.

El objetivo que trata de alcanzar este tema es permitir el desarrollo de programas completos desde el principio. Estos programas se podrán realizar como prácticas con el computador de manera inmediata y directa utilizando un compilador de C/C++.

## 2.1 Lenguaje C±

El lenguaje de programación **C±** (léase C-más-menos), que se utilizará a lo largo de todo este libro para introducir los diferentes conceptos de programación, está constituido por un subconjunto del vocabulario de los lenguajes C y C++. Por lo tanto, cualquier programa escrito en el *lenguaje* **C±** se podrá editar, compilar y ejecutar en un entorno de desarrollo para C++ que incluya como subconjunto al lenguaje C. Conviene señalar que los ficheros fuente que contengan los programas **C±** deberán tener extensión `.cpp` como si fueran programas en C++.

El objetivo fundamental de utilizar el lenguaje **C±** es la introducción de los conceptos fundamentales de programación de una manera progresiva, sistemática y sin ambigüedades con el fin de que se adquiera una buena metodología de programación. Lamentablemente C o C++ no fueron diseñados para la formación de programadores y disponen de ciertas estructuras que conceptualmente son poco rigurosas y que por ello no forman parte de **C±**.

El lenguaje **C±** se irá presentando de manera simultánea a la introducción de los conceptos según se avance en el curso. La presentación de cada nuevo elemento de **C±** se realizará formalmente mediante la notación BNF (ver siguiente epígrafe de este tema). En el apéndice A de este libro se recopila la descripción formal en BNF de la sintaxis completa del lenguaje **C±**.

En cada tema sólo se utilizarán aquellos elementos del lenguaje **C±** que ya hayan sido presentados. Dado el enfoque metodológico de la asignatura, cualquier programa o práctica que no se realice en el lenguaje **C±** y siguiendo las pautas marcadas a lo largo del libro se considerará incorrecto. Aunque el exigir restricciones como éstas pueda parecer de carácter solo formativo o estrictamente académico, el poner ciertas restricciones para la codificación en determinados lenguajes de programación resulta una práctica bastante habitual en el desarrollo de software a nivel industrial.

Siguiendo las pautas de buenas prácticas de ingeniería de software, cualquier empresa o equipo de desarrollo de software debe disponer antes del inicio de cada desarrollo de un *Manual de Estilo*. Para lograr la adecuada claridad, homogeneidad y mantenibilidad de los programas, en el *Manual de Estilo*, se establecen prohibiciones expresas de uso de algunas estructuras del lenguaje de programación empleado, el formato de escritura de cada sentencia, recomendaciones de uso de los distintos elementos del lenguaje (constantes, variables, tipos y subprogramas) y otros muchos aspectos. Para garantizar que todo el desarrollo sigue estas pautas establecidas, una o varias personas del departamento de calidad son las encargadas del garantizar la calidad requerida de todos los programas y para ello tienen la potestad de exigir las correcciones o modificaciones que consideren necesarias a los programadores.

## 2.2 Notación BNF

Un lenguaje de programación sigue unas reglas gramaticales similares a las de cualquier idioma humano, aunque más estrictas. Para la definición formal de dichas reglas sintácticas utilizaremos la *notación BNF* (Backus-Naur Form) basada en la descripción de cada elemento gramatical en función de otros más

sencillos, según determinados esquemas o construcciones. Cada uno de estos esquemas se define mediante una *regla de producción.*

Estas reglas sobre cómo han de escribirse los elementos del lenguaje en forma de símbolos utilizan a su vez otros símbolos, que se denominan *metasímbolos.* Son los siguientes:

**::=** Metasímbolo de definición. Indica que el elemento a su izquierda puede desarrollarse según el esquema de la derecha.

**|** Metasímbolo de alternativa. Indica que puede elegirse uno y sólo uno de los elementos separados por este metasímbolo.

**{ }** Metasímbolos de repetición. Indican que los elementos incluidos dentro de ellos se pueden repetir cero o más veces.

**[ ]** Metasímbolos de opción. Indican que los elementos incluidos dentro de ellos pueden ser utilizados o no.

**( )** Metasímbolos de agrupación. Agrupan los elementos incluidos en su interior.

Estos metasímbolos se escriben con el tipo de letra especial indicado para distinguirlos de los paréntesis, corchetes, etc. que forman parte del lenguaje **C±**. También se emplearán distintos estilos de letra para distinguir los elementos simbólicos siguientes:

*Elemento_no_terminal* Este estilo se emplea para escribir el nombre de un elemento gramatical que habrá de ser definido por alguna regla. Cualquier elemento a la izquierda del metasímbolo **::=** será no terminal y aparecerá con este estilo.

`Elemento_terminal` Este estilo se emplea para representar los elementos que forman parte del lenguaje **C±**, es decir, que constituyen el texto de un programa. Si aparecen en una regla deberán escribirse exactamente como se indica.

## 2.3 Valores y tipos

El computador, como máquina de tratamiento de información, manipula diferentes datos. Un *dato* es un elemento de información que puede tomar un *valor* entre varios posibles. Si un dato tiene siempre necesariamente un valor fijo, diremos que es una *constante.*

Los valores de los datos pueden ser de diferentes clases. En general un dato sólo puede tomar valores de una clase. Por ejemplo, la estatura de una persona no puede tomar el valor "Felipe", ni el nombre de una persona puede ser "175".

En programación a las distintas clases de valores se les denomina *tipos*. Un dato tiene asociado un tipo, que representa la clase de valores que puede tomar. Por ejemplo, son tipos diferentes:

- Los números enteros.
- Los días de la semana.
- Los meses del año.
- Los títulos de libros.
- ... etc. ...

Es importante destacar que el concepto de tipo es algo abstracto, e independiente de los símbolos concretos que se empleen para *representar* los valores. Por ejemplo, aunque podemos representar los meses del año mediante números enteros de 1 a 12, los meses no son números enteros, pues no tiene sentido, por ejemplo, sumar Enero (1) y Marzo (3) para obtener Abril (4).

Con más precisión se habla de *tipos abstractos de datos*, que identifican tanto el conjunto de valores que pueden tomar los datos de ese tipo como las operaciones significativas que pueden hacerse con dichos valores.

En la comunicación humana usamos habitualmente dos grandes clases de valores: los *números* y los *textos*. Los lenguajes de programación llevan incluidas formas de representación concretas de estas clases de valores, que se traducen en la existencia de tipos de datos predefinidos, ya incorporados al lenguaje, y que pueden usarse, en su caso, para representar también valores de otros nuevos tipos de datos definidos por el programador. Aunque en la práctica los números han de escribirse externamente en forma de texto para poder ser leídos por las personas, desde el punto de vista abstracto son valores de tipos diferentes a los de los caracteres que los representan.

## 2.4 Representación de valores constantes

Uno de los objetivos de los lenguajes de programación es evitar las ambigüedades o imprecisiones que existen en los lenguajes humanos. Por ejemplo, la representación de valores numéricos en los países anglosajones se realiza separando por comas (,) los millares. Así, trescientos cuarenta y ocho mil quinientos treinta y seis se representa de la siguiente manera:

348,536

Sin embargo, nosotros utilizamos la coma para separar la parte entera de la parte decimal de un número no entero. Por lo tanto, la interpretación con esta regla del número anterior sería: trescientos cuarenta y ocho con quinientas treinta y seis milésimas.

A continuación se indican las reglas particulares de **C±** para la representación de valores básicos, tanto numéricos como de texto.

### 2.4.1 Valores numéricos enteros

Los valores enteros representan un número exacto de unidades, y no pueden tener parte fraccionaria. Un *valor entero* se escribe mediante una secuencia de uno o más dígitos del 0 al 9 sin separadores de ninguna clase entre ellos y precedidos opcionalmente de los símbolos más (+) o menos (−). Son enteros válidos los siguientes:

```
2
+56
0
-234567745
1000000000
```

Sin embargo, no son valores enteros válidos los siguientes:

| | |
|---|---|
| `123,234` | No se pueden usar comas |
| `22.56` | No se pueden usar puntos |
| `13E5` | No se pueden usar letras |

Usando la notación BNF podemos representar de manera precisa las reglas para escribir estos valores:

*Valor_entero* **::=** **[** + **|** - **]** *Secuencia_dígitos*

*Secuencia_dígitos* **::=** *Dígito* **{** *Dígito* **}**

*Dígito* **::=** 0 **|** 1 **|** 2 **|** 3 **|** 4 **|** 5 **|** 6 **|** 7 **|** 8 **|** 9

▌NOTA: El lenguaje **C±** como derivado de C/C++ considera que cuando el valor entero comienza por un primer dígito 0 se está escribiendo en base 8 (octal) en lugar de en base 10 (decimal). Así, el valor númerico 020 es un número octal que equivale a 16 en decimal. En este curso no se hace uso de los valores octales y carece de sentido poner ceros a la izquierda de un valor numérico. En cualquier caso, el compilador de C/C++ da un error si al escribir un valor octal se utilizan los dígitos 8 ó 9.

### 2.4.2 Valores numéricos reales

Los valores numéricos reales permiten expresar cualquier cantidad, incluyendo fracciones de unidad. Se pueden representar de dos maneras distintas: en la notación decimal habitual, o en la notación científica. En la notación decimal habitual un *valor real* se escribe con una parte entera terminada siempre por un punto (`.`), y seguida opcionalmente por una secuencia de dígitos que constituyen la parte fraccionaria decimal. De acuerdo con ello son valores reales válidos los siguientes:

```
5.
-0.78
+234.53
0.0000000034
1234.000
```

En la notación científica un número real se escribe como una *mantisa*, que es un número real en la notación decimal habitual, seguida de un factor de escala que se escribe como la letra `E` seguida del exponente entero de una potencia de 10 por la que se multiplica la mantisa. Son valores reales válidos en notación científica:

| | |
|---|---|
| `-23.2E+12` | equivalente a $-23.2\times10^{12}$ |
| `14567.823E4` | equivalente a $14567.823\times10^{4}$ |
| `126.E-34` | equivalente a $126\times10^{-34}$ |

Sin embargo, no son valores reales válidos los siguientes:

| | |
|---|---|
| `4,78` | No se pueden usar comas |
| `56.7F-56` | No se puede usar la letra `F` |

A diferencia de los valores enteros, un mismo valor real puede tener muy diversas representaciones válidas. Por ejemplo, todas las representaciones siguientes corresponden al mismo valor:

```
45.6      456.E-1      4.56E+1      45.60E+0      456000.00E-4
```

Las reglas anteriores, expresadas en notación BNF son:

*Valor_real* **::=** *Valor_entero* . **[** *Secuencia_dígitos* **]** **[** *Escala* **]**

*Escala* **::=** E *Valor_entero*

### 2.4.3 Caracteres

Además de los valores numéricos enteros o reales, empleados para la realización de cálculos numéricos, los lenguajes de programación nos deben permitir representar valores correspondientes a los caracteres de un texto, y que están disponibles en cualquier teclado, pantalla o impresora.

Dentro del texto de un programa en **C±** el *valor de un carácter* concreto se escribe poniendo dicho carácter entre apóstrofos ('). Ejemplos de valores de caracteres son los siguientes:

```
'a'   'Ñ'   '7'   '?'   '.'   ' '   '}'
'A'   'ñ'   '5'   '!'   ';'   '{'   '"'
```

Es interesante hacer las siguientes observaciones:

- el espacio en blanco (`' '`) es un carácter válido como los demás
- hay que distinguir entre un valor entero de un dígito (p.ej. `7`) y el carácter correspondiente a dicho dígito (p.ej. `'7'`)

La colección o *juego de caracteres* (*charset*) que pueden manipularse en un programa depende de la máquina que se esté usando. Sólo se pueden representar de la forma indicada (escribiéndolos entre apóstrofos) aquellos caracteres que tengan asociado un símbolo gráfico (letra, dígito, signo de puntuación, etc.) que pueda introducirse en el texto del programa. Otros caracteres definidos, tales como los *caracteres de control*, que no tienen símbolo gráfico, se representan mediante una *secuencia de escape* con la siguiente notación:

| | |
|---|---|
| `'\n'` | Salto al comienzo de una nueva línea de escritura |
| `'\r'` | Retorno al comienzo de la misma línea de escritura |
| `'\t'` | Tabulación |
| `'\''` | Apóstrofo |
| `'\\'` | Barra inclinada |
| `'\f'` | Salto a una nueva página o borrado de pantalla |

### 2.4.4 Cadenas de caracteres (*strings*)

Es frecuente que los caracteres no se utilicen de forma aislada, sino formando palabras o frases. Una *cadena de caracteres* (en inglés *string*) se escribe como una secuencia de caracteres incluidos entre comillas ("). Por ejemplo:

```
"Palabra"
"Este texto es una cadena de caracteres"
"&"
"El resultado de A+B es: "
"Incluir entre 'apóstrofos' el texto"
"Conteste \"Si\" o \"No\""
"¿Año de fabricación?"
""
```

Conviene observar que:

- si una cadena incluye comillas en su interior se escribirá mediante `\"`
- no hay que confundir un valor de tipo carácter (`'x'`) con una cadena del mismo único carácter (`"x"`). La distinción se produce por el delimitador utilizado comillas (`"`) para una cadena y apóstrofo (`'`) para un carácter
- es posible definir una cadena vacía que no contenga ningún carácter, como en el último ejemplo

Una cadena puede contener cualquier número de caracteres y puede incluir cualquier carácter alfabético o de puntuación que sea representable dentro del texto del programa. Aquí se aplican las mismas observaciones que se han hecho antes respecto al juego de caracteres particular de cada máquina.

## 2.5 Tipos predefinidos

Las diferentes formas de representación de valores constantes presentadas en los apartados anteriores distinguen ya varias clases de datos, pero que no llegan a ser tipos en sí mismos. En efecto, tal como vamos a ver a continuación, dentro de una misma clase de valores pueden distinguirse varios tipos diferentes, tanto a nivel de *tipos predefinidos* en el lenguaje, como de *tipos definidos* por el programador.

Recordaremos que un tipo de datos define:

1. Una colección de valores posibles
2. Las operaciones significativas sobre ellos

En el lenguaje **C±** hay cuatro tipos de datos predefinidos, que se designan con los nombres `int`, `float`, `char`, `bool`, así como mecanismos para definir nuevos tipos a partir de ellos. En los lenguajes C y C++ hay tipos predefinidos adicionales. En las secciones siguientes se describen los tipos predefinidos fundamentales, excepto el tipo `bool`, que se describe más adelante.

### 2.5.1 El tipo entero (int)

Los valores de este tipo son los valores numéricos enteros positivos y negativos. Como tipo abstracto su definición coincide con el concepto matemático de los números enteros. Sin embargo, dado el carácter físico de los computadores, el rango de valores nunca podrá ser infinito como se establece en el concepto matemático. En cada caso el rango de valores del *tipo* `int` depende de la *plataforma* (combinación de: procesador, sistema operativo y compilador) que se esté utilizando. En general se corresponde con el rango de valores que pueden manipularse con instrucciones básicas del lenguaje de máquina y viene a ser simétrico en torno al valor cero. Dentro de dicho rango la representación de cualquier valor es exacta. Son rangos comunes los siguientes:

| Tamaño de palabra | Rango de valores enteros |
|---|---|
| 16 bits | -32.768 ... 0 ... 32.767 |
| 32 bits | -2.147.483.648 ... 0 ... 2.147.483.647 |
| 64 bits | -9.223.372.036.854.775.808<br>... 0 ...<br>9.223.372.036.854.775.807 |

Estos rangos obedecen a que los computadores suelen emplear la codificación en base 2 de los valores enteros. Para el signo del número se utiliza un bit, quedando, por tanto, 15, 31 ó 63 para el valor absoluto:

$$2^{15} = 32.768$$
$$2^{31} = 2.147.483.648$$
$$2^{63} = 9.223.372.036.854.775.808$$

▌NOTA: Para facilitar la escritura de programas que tengan en cuenta la limitación particular de rango existente en cada caso, C y C++ permiten hacer referencia al valor mínimo mediante el nombre simbólico `INT_MIN`, y al valor máximo mediante `INT_MAX`. El rango admisible será, por tanto: `INT_MIN` ... 0 ... `INT_MAX`. Estos nombres están definidos en el módulo `limits` de la librería estándar de C (cabecera `<limits.h>`).

Asociadas al tipo `int` están las operaciones que se pueden realizar con los valores de este tipo. Las operaciones predefinidas entre valores enteros son las operaciones aritméticas básicas, que se realizan entre enteros y devuelven como resultado valores enteros. Para invocar estas operaciones se dispone de los siguientes símbolos de operación u *operadores*:

| | | |
|---|---|---|
| `+` | Suma de enteros | `a + b` |
| `-` | Resta de enteros | `a - b` |
| `*` | Multiplicación de enteros | `a * b` |
| `/` | División de enteros | `a / b` |
| `%` | Resto de la división | `a % b` |
| `+` | Identidad de un entero | `+ a` |
| `-` | Cambio de signo de un entero | `- a` |

Siguiendo la representación aritmética habitual, los símbolos `+` y `-` tienen un doble significado, según se usen como operadores infijos entre 2 operandos o como operadores prefijos con un único operando.

El operador `/` realiza la división entre dos números enteros y obtiene como resultado el cociente entero truncado al valor más próximo a cero. Cuando el divisor es cero se obtiene como resultado un error. Por ejemplo:

| **Operación** | **Resultado** |
|---|---|
| `10 / 3` | 3 |
| `(-20) / (-7)` | 2 |
| `(-15) / 4` | -3 |
| `17 / (-3)` | -5 |
| `34 / 0` | *Error* |

El operador `%` obtiene el resto de la división de enteros realizada con `/`. Por ejemplo:

| **Operación** | **Resultado** |
|---|---|
| `10 % 3` | 1 |
| `(-20) % (-7)` | -6 |
| `(-15) % 4` | -3 |
| `17 % (-3)` | 2 |
| `34 % 0` | *Error* |

Entre los operadores `/` y `%` se cumple la regla aritmética habitual:

$$\textit{Dividendo} = \textit{Divisor} \times \textit{Cociente} + \textit{Resto}$$

que en **C±** se expresaría así:

```
a  =  b * (a/b) + (a%b)
```

Cuando se realiza una operación con enteros se debe tener en cuenta el rango de valores disponible en la plataforma que se esta utilizando. Si se produce un resultado fuera del rango disponible se producirá un error. En algunos casos este tipo de errores no se indica y puede ser difícil su detección. Por ejemplo, para enteros de 16 bits con un rango entre -32.768 y 32.767, se obtienen los siguientes resultados:

| Operación | Resultado |
|---|---|
| `234 + 89` | 323 |
| `345 * 97` | *Error* |
| `214 * (-203)` | *Error* |
| `15456 + 18678` | *Error* |
| `(-20) - 32750` | *Error* |

### 2.5.2 El tipo real (`float`)

Con el *tipo* `float` se trata de representar en el computador los valores numéricos reales positivos y negativos. Sin embargo, al contrario que en caso del tipo `int`, esta representación puede no ser exacta. Además, dado que la capacidad de los computadores es limitada, la representación sólo se puede considerar válida dentro de un rango, de forma semejante a como sucede con los enteros.

Tanto el rango como la precisión dependen de la plataforma concreta utilizada. Dentro de dicho rango para algunos valores concretos es posible una representación exacta. Sin embargo, dado el carácter discreto de los datos que siempre se manejan en un computador, nunca será posible una representación exacta de valores tales como los valores irracionales $\pi$ o $e$ o, en general, de valores cuya precisión sea superior a la disponible en la plataforma que estemos utilizando. En estos casos se manejan valores aproximados.

Los valores reales se suelen representar internamente de forma equivalente a la notación científica, con una mantisa y un factor de escala. El rango de valores representables está limitado tanto para valores grandes como pequeños. Los valores más pequeños que un límite dado se confunden con el cero. Al igual que en el caso de los valores enteros, el rango y precisión de los valores reales puede cambiar de una plataforma a otra. Algunos de los rangos habituales son los siguientes:

| Tamaño de palabra y precisión | Rango de valores reales |
|---|---|
| 32 bits; 6 cifras decimales | -3.4E+38 ... -1.2E-38<br>0<br>+1.2E-38 ... +3.4E+38 |
| 64 bits; 15 cifras decimales | -1.7E+308 ... -2.3E-308<br>0<br>+2.3E-308 ... +1.7E+308 |

Estos rangos dependen del número concreto de bits y de la codificación que se emplean para la mantisa y el exponente del valor `float`. En el caso de valores

representados con 32 bits no existe ningún valor intermedio entre $-1,2\times10^{-38}$ y el valor 0, ni tampoco entre 0 y $1,2 \times 10^{-38}$ (y análogamente para valores en 64 bits).

Asociadas al tipo `float` están las operaciones que se pueden realizar con él. Las operaciones entre valores reales son las operaciones aritméticas básicas, que se realizan entre reales y devuelven como resultado valores reales. Los correspondientes operadores son los siguientes:

| | | |
|---|---|---|
| + | Suma de reales | `a + b` |
| - | Resta de reales | `a - b` |
| * | Multiplicación de reales | `a * b` |
| / | División de reales | `a / b` |
| + | Identidad de un real | `+ a` |
| - | Cambio de signo de un real | `- a` |

Los símbolos empleados para estos operadores son los mismos que para los operadores enteros. Sin embargo, en todos los casos son operadores distintos de los operadores enteros. Las operaciones entre reales dan como resultado un real con la precisión de la plataforma. Así, para valores reales no se cumple siempre exactamente la relación básica:

$$\texttt{(a / b) * b} = \texttt{a}$$

Es importante tener en cuenta imprecisiones como ésta cuando los cálculos sean más complejos y se puedan acumular errores. Ejemplos de operaciones entre valores reales son las siguientes:

| **Operación** | **Resultado** |
|---|---|
| `10.5 / 0.2` | 52.5 |
| `-20.6 * 2.4` | -49.44 |
| `-15.4E2 + 450.0` | -1090.0 |
| `23.4 - 2E-1` | 23.2 |

La representación sólo aproximada de los valores reales se pone de manifiesto si tratamos de expresar con más precisión de la realmente existente el resultado de una operación. Por ejemplo:

| **Operación** | **Resultado** |
|---|---|
| `10.0 / 3.0` | `3.3333332538604736E+0` |

En este caso concreto el valor es inexacto a partir de la 7ª cifra decimal. En cada plataforma se podrá obtener un resultado ligeramente diferente.

### 2.5.3 El tipo carácter (char)

Para comprender bien el manejo de valores de *tipo carácter* en un computador es necesario conocer cómo se definen y representan esos valores de caracteres. Cada carácter no se representa internamente como un dibujo (el *glifo* del carácter), sino como un valor numérico entero que es su código. La colección concreta de caracteres y sus códigos numéricos se establecen en una tabla (*charset*) que asocia a cada carácter el código numérico (*codepoint*) que le corresponde.

Dependiendo del número de bits reservado para representar el código de cada carácter podremos tener tablas más o menos amplias. Algunas tablas de caracteres de amplio uso son:

| **Tabla (*charset*)** | **Tamaño del carácter** | **Repertorio de caracteres** |
|---|---|---|
| ASCII | 7 bits | Letras inglesas mayúsculas y minúsculas. Algunos signos de puntuación y códigos de control. |
| ISO-8859-1 (llamado también Latin-1) | 8 bits | Lo anterior, más letras con acentos y nuevos signos de puntuación |
| UNICODE-BMP (Basic Multilingual Plane) | 16 bits | Incluye además los alfabetos griego, cirílico, árabe, chino/japonés/coreano, signos matemáticos, etc. |
| UNICODE completo | 32 bits | Incluye la práctica totalidad de caracteres utilizados en cualquier idioma o notación textual existente en nuestro mundo actual. |

Las tablas mencionadas son compatibles entre sí en el sentido de que cada una de ellas incluye la anterior, manteniendo los códigos numéricos de los caracteres. Lamentablemente la compatibilidad no se extiende a otras muchas tablas de caracteres de amplio uso. Por ejemplo, otras tablas de caracteres de 8 bits muy conocidas son:

| Tabla (*charset*) | Tamaño del carácter | Repertorio de caracteres |
|---|---|---|
| ISO-8859-7 | 8 bits | Repertorio ASCII más el alfabeto griego |
| ISO-8859-15 | 8 bits | Repertorio Latin-1 revisado. Incluye el símbolo de Euro |
| IBM-PC-437 (original del sistema operativo MS-DOS) | 8 bits | Repertorio ASCII más simbolos semigráficos y letras con acentos, pero con códigos distintos de Latin-1 |
| IBM-PC-850 (usado también en MS-DOS) | 8 bits | Casi el mismo repertorio de Latin-1 pero con códigos diferentes, y diferentes también de IBM-PC-437. |
| Windows-1252 (usado en los sistemas operativos MS-Windows) | 8 bits | Coincide con Latin-1 excepto en un rango de 32 códigos de Latin-1 que repiten códigos de control |

En **C±** (como en C/C++) los valores del tipo **char** ocupan 8 bits e incluyen el repertorio ASCII. Además incluyen otros caracteres no-ASCII que dependen de la tabla de caracteres establecida. En los ejemplos de este libro asumiremos que se dispone de los caracteres comunes a Latin-1 y Windows-1252. Por lo tanto la colección de valores del tipo **char** incluye caracteres alfabéticos, numéricos, de puntuación y caracteres de control.

Como ya se ha dicho, en el texto de un programa se pueden escribir los valores de los caracteres, bien directamente, o mediante una secuencia de escape, p.ej. para los caracteres de control. También se puede representar cualquier carácter mediante la notación **char**($x$) siendo $x$ el código del carácter. Por ejemplo, en ASCII:

| | |
|---|---|
| **char(10)** | Salto al comienzo de una nueva línea. Posición 10ª |
| **char(13)** | Retorno al comienzo de la misma línea. Posición 13ª |
| **char(65)** | Letra A mayúscula. Posición 65ª |

En sentido inverso, el código numérico de un determinado carácter $c$ se expresa como `int(`$c$`)`. Por ejemplo:

| | |
|---|---|
| `int('A')` | 65 (65ª posición de la tabla ASCII) |
| `int('Z')` | 90 (90ª posición de la tabla ASCII) |

De forma inmediata se puede decir que, para cualquier carácter $c$, cuyo código sea $x$, se cumplirá que:

```
char(int(c)) = c
int(char(x)) = x
```

Además conviene saber que la tabla ASCII posee las siguientes características:

- Los caracteres correspondientes a las letras mayúsculas de la 'A' a la 'Z' están ordenados en posiciones consecutivas y crecientes según el orden alfabético.
- Los caracteres correspondientes a las letras minúsculas de la 'a' a la 'z' están ordenados en posiciones consecutivas y crecientes según el orden alfabético.
- Los caracteres correspondientes a los dígitos del '0' al '9' están ordenados en posiciones consecutivas y crecientes.

Esto facilita el obtener por cálculo el valor numérico equivalente al carácter de un dígito decimal, o la letra mayúscula correspondiente a una minúscula o viceversa.

En C (y en **C±**) se puede usar también el módulo de librería `ctype` (cabecera `<ctype.h>`), que facilita el manejo de diferentes clases de caracteres. Este módulo incluye funciones tales como:

| | |
|---|---|
| **isalpha**( c ) | Indica si c es una letra |
| **isascii**( c ) | Indica si c es un carácter ASCII |
| **isblank**( c ) | Indica si c es un carácter de espacio o tabulación |
| **iscntrl**( c ) | Indica si c es un carácter de control |
| **isdigit**( c ) | Indica si c es un dígito decimal (0-9) |
| **islower**( c ) | Indica si c es una letra minúscula |
| **isspace**( c ) | Indica si c es espacio en blanco o salto de línea o página |
| **isupper**( c ) | Indica si c es una letra mayúscula |
| **tolower**( c ) | Devuelve la minúscula correspondiente a c |
| **toupper**( c ) | Devuelve la mayúscula correspondiente a c |

▌NOTA: El concepto de función se introduce en el tema 7, y el de módulo en el tema 14.

## 2.6 Expresiones aritméticas

Una *expresión aritmética* representa un cálculo a realizar con valores numéricos (más adelante se verán expresiones que utilizan también valores de otros tipos). Una expresión aritmética es una combinación de *operandos* y *operadores*.

Para indicar el orden en que se quieren realizar las operaciones parciales se pueden utilizar paréntesis. Si no se utilizan paréntesis el orden de las operaciones depende de una jerarquía entre los operadores empleados, que para los operadores numéricos es la siguiente:

| | |
|---|---|
| 1º Operadores multiplicativos: | `* / %` |
| 2º Operadores aditivos: | `+ -` |

Dentro del mismo nivel las operaciones se ejecutan en el orden en que están escritas en la expresión aritmética de izquierda a derecha.

Si una expresión va precedida del signo más o menos, se entiende que solamente le afecta al primer operando. Si se quiere que afecte a toda la expresión, ésta deberá incluirse entre paréntesis.

Ejemplos de expresiones entre datos enteros son las siguientes:

| **Expresión** | **Resultado** |
|---|---|
| `5 * 30 + 5` | 155 |
| `334 / 6 % 4 * 5` | 15 |
| `-5 * 10 % 3 / 2` | -1 |

Cuando la complejidad de la expresión puede dar lugar a posibles errores de interpretación, es preferible utilizar paréntesis para clarificar cuál es el cálculo exacto que se quiere realizar. Así las expresiones anteriores son equivalentes a las siguientes:

```
(5 * 30) + 5
((334 / 6) % 4) * 5
(((-5) * 10) % 3) / 2
```

Igualmente, ejemplos de expresiones entre valores reales son las siguientes:

| **Expresión** | **Resultado** | **Expresión equivalente** |
|---|---|---|
| `35.3 * 5.1 / 7.6 - 4.5` | 19.18816 | `((35.3*5.1)/7.6)-4.5` |
| `-23.1 / 6.2 * 5.4 / 2.4` | -8.38306 | `(((-23.1)/6.2)*5.4)/2.4` |

Aunque se represente por los mismos símbolos, los operadores aritméticos para valores reales y enteros son en realidad diferentes. Así, si se mezclaran en una misma expresión valores de tipos diferentes, las expresiones aritméticas son completamente ambiguas. Por ejemplo en las siguientes operaciones:

| | |
|---|---|
| `33.7 / 5` | ¿La división a realizar es entera o real? |
| `33 / 5.3` | ¿La división a realizar es entera o real? |
| `25 * 3.5` | ¿La multiplicación a realizar es entera o real? |

Evidentemente, los resultados serán diferentes según el tipo de operación que se realice. Además no queda claro si el resultado que se pretende obtener es un valor entero o real. Para poder realizar estas operaciones combinadas es necesario que previamente se realice una conversión de la representación de los datos al tipo adecuado. La representación real de un dato entero se indica de la siguiente manera:

`float(45)` Representa el valor numérico 45.0 con tipo `float`

De forma similar la representación entera de un dato real (correspondiente a la parte entera, truncando el valor) se consigue de la siguiente forma:

`int(34.7)` Representa el valor numérico 34 con tipo `int`

Por tanto, si queremos obtener un resultado entero, las operaciones entre enteros y reales anteriormente indicadas se tienen que realizar de la siguiente manera:

| Expresión | Resultado | Tipo |
|---|---|---|
| `int(33.7) / 5` | 6 | `int` |
| `33 / int(5.3)` | 6 | `int` |
| `25 * int(3.5)` | 75 | `int` |

Si el resultado deseado es un valor real, es necesario realizar previamente las conversiones a real de los operandos enteros y lógicamente obtendremos resultados completamente distintos:

| Expresión | Resultado | Tipo |
|---|---|---|
| `33.7 / float (5)` | 6.74 | `float` |
| `float(33) /5.3` | 6.22642 | `float` |
| `float(25) * 3.5` | 87.5 | `float` |

El lenguaje **C±** permite la ambigüedad que supone la mezcla de tipos de datos diferentes en la misma expresión sin exigir una conversión explícita. Para salvar la ambigüedad, **C±** utiliza el convenio de C/C++ de convertir previamente de manera automática todos los valores de una misma expresión al tipo del valor con mayor rango y precisión. Por tanto, el resultado siempre se obtendrá también en el mayor rango y precisión utilizado en la expresión. El desconocimiento de esta regla implícita, puede dar lugar a que el resultado de una expresión aritmética sea completamente inesperado. Para evitar esta situación, en el *Manual de Estilo* para la realización de programas en esta asignatura es obligatorio que se realice siempre una conversión explícita de tipos.

## 2.7 Operaciones de escritura simples

El objetivo de un programa es obtener unos resultados. Estos resultados deben ser emitidos al exterior del computador a través de un dispositivo de salida de datos: impresora, pantalla, trazador (*plotter*), línea de comunicaciones, etc. Las acciones que envían resultados al exterior se llaman, en general, *operaciones de escritura*, con independencia de que se trate de una impresión en papel, o la simple visualización en pantalla, o la grabación de los datos en un soporte donde queden registrados, o su envío a otro equipo remoto.

Existe una gran variedad de dispositivos periféricos, que se diferencian mucho en los detalles de su manejo. Para simplificar la escritura de resultados los lenguajes de programación prevén sentencias de escritura apropiadas para ser usadas con cualquier tipo de dispositivo, facilitando la tarea de programación al especificar la escritura de resultados de una manera uniforme, con independencia de las particularidades del dispositivo físico que se utilice en cada caso.

Al diseñar un lenguaje de programación se puede optar por usar sentencias o instrucciones especiales para ordenar la escritura de resultados, o bien ordenar la escritura del resultado con las mismas sentencias generales que se empleen para invocar operaciones definidas por el usuario. Los primeros lenguajes de programación solían emplear la primera alternativa. Los lenguajes más modernos utilizan con preferencia la segunda, que simplifica la complejidad del lenguaje en sí, a costa de permitir a veces una cierta variación en las operaciones de escritura entre diferentes versiones del lenguaje.

El lenguaje **C±** adopta también la segunda alternativa que es la utilizada en C/C++. Las operaciones de escritura se definen como procedimientos (ver tema 7), que se invocan escribiendo el nombre de la operación, seguido de una serie de valores o argumentos entre paréntesis. Estos *procedimientos* están definidos en *módulos de librería* (ver tema 14) disponibles de antemano.

En todas las versiones de C/C++ deben estar disponibles ciertos módulos estándar con la definición de operaciones de escritura normalizadas. En este apartado describiremos una operación disponible en el módulo llamado `stdio`.

### 2.7.1 El procedimiento `printf`

Este procedimiento pertenece al módulo `stdio` (cabecera `<stdio.h>`). La forma más sencilla de invocarlo es escribir:

`printf(` *cadena-de-caracteres* `);`

▌NOTA: Esta forma sencilla sólo es válida si la cadena de caracteres a escribir no contiene el carácter `%`.

El procedimiento **printf** escribe en la pantalla del computador la cadena de caracteres. Por ejemplo, para cada una de las siguientes operaciones de escritura se obtiene el resultado que se muestra a su derecha. Para visualizar con detalle el resultado se ha utilizado el símbolo '·' para representar el carácter de espacio en blanco.

| Operación de escritura | Resultado en pantalla |
|---|---|
| `printf( "En un lugar de " );` | `En·un·lugar·de·` |
| `printf( "¿Año de nacimiento?" );` | `¿Año·de·nacimiento?` |

Si lo que se quiere escribir es la representación como texto de una serie de valores de cualquier tipo de los vistos hasta el momento (enteros, reales, caracteres, etc.), habrá que usar la forma general de la orden `printf`:

`printf(` *cadena-con-formatos, valor1, valor2, ... valorN* `);`

Una cadena de caracteres con formatos deberá incluir en su interior una especificación de formato por cada valor que se quiera insertar. La forma más simple de especificar un formato es mediante `%`*x*, es decir, usando el carácter fijo `%` seguido de una letra de código que indica el tipo de formato a aplicar. Algunos códigos de formato habituales son:

| Código | Nemotécnico (*inglés*) | Tipo de valor |
|---|---|---|
| d | **d***ecimal* | entero |
| f | **f***ixed point* | real |
| e | **e***xponential* | real con notación exponencial |
| g | **g***eneral* | real con/sin notación exponencial |
| c | **c***haracter* | un carácter |
| s | **s***tring* | una cadena de caracteres |

Por ejemplo, para cada una de las siguientes operaciones de escritura se obtiene el resultado que se muestra a su derecha:

| Operación de escritura | Resultado |
|---|---|
| `printf( "%d", 120 / 12 );` | `10` |
| `printf( "Datos:%d#%d", 23*67, -50 );` | `Datos:1541#-50` |
| `printf( "Datos: %d # %d", 23*67, -50 );` | `Datos:·1541·#·-50` |

Como se puede apreciar en los ejemplos estos formatos simples usan sólo el número de caracteres estrictamente necesarios para escribir el valor de cada dato, sin añadir espacios en blanco. Si se quiere separar con espacios unos valores de otros, entonces hay que incluirlos en el formato.

Otra forma de conseguir espacios en los resultados es indicar explícitamente cuántos caracteres debe ocupar el valor de cada dato escrito. Esto se hace poniendo el número de caracteres entre el símbolo de % y el código del formato. Ejemplos:

| Operación de escritura | Resultado |
|---|---|
| `printf( "%5d", 120/12 );` | `···10` |
| `printf( "Datos:%7d#%5d", 23*6, -50 );` | `Datos:····138#··-50` |
| `printf( "%3d", 1000*34 );` | `34000` |

Cuando el número de caracteres indicado es insuficiente para representar completamente el valor, como ocurre en el último ejemplo, se utilizan tantos caracteres como sean necesarios para que el resultado aparezca completo.

Además, cuando se utiliza un formato **f**, **e** ó **g** se puede especificar también el número de cifras decimales que se deben escribir después del punto decimal. Por ejemplo:

| Operación de escritura | Resultado |
|---|---|
| `printf( "%10.3f", 1.2 );` | `·····1.200` |
| `printf( "%10.4e", 23.1*67.4 );` | `0.1557E+04` |
| `printf( "%15.3g", -50.6E-6 );` | `······-0.506E-04` |

Salvo que se indique otra cosa, los resultados obtenidos mediante sucesivas sentencias de escritura van apareciendo en el dispositivo de salida uno tras otro en la misma línea de texto. Por ejemplo, las siguientes sentencias de escritura:

```
printf( "Area = " );
printf( "%10.4f", 24.45 );
printf( "Mi ciudad es Avila" );
printf( "Descuento: " );
printf( "%5.2d", 12.5 );
printf( "%c", '%' );
```

producen el resultado siguiente (se prescinde ya del símbolo '·' para representar un espacio en blanco):

```
Area =    24.4500Mi ciudad es AvilaDescuento: 12.50%
```

Para escribir resultados en varias líneas de texto habrá que recordar que dentro de una cadena se pueden incluir caracteres especiales mediante secuencias de escape. Más concretamente, si queremos dar por terminada una línea de resultados y pasar a escribir en la siguiente bastará con incluir en el punto adecuado la secuencia de escape **\n**. Por ejemplo, si se modifican ligeramente las anteriores operaciones de escritura:

```
printf( "Area = " );
printf( "%10.4f\n", 24.45 );
printf( "Mi ciudad es Avila\n" );
printf( "Descuento: " );
printf( "%5.2d", 12.5 );
printf( "%c\n", '%' );
```

se obtendrá como resultado:

```
Area =    24.4500
Mi ciudad es Avila
Descuento: 12.50%
```

## 2.8 Estructura de un programa completo

Con los elementos introducidos hasta este momento pueden formarse ya programas completos. Sólo falta indicar cuál es la estructura global del programa.

Un programa en **C±** se engloba dentro de una estructura principal o `main()`. Un ejemplo de programa muy sencillo es el siguiente:

```
/** Programa: Hola */
/* Este programa escribe "Hola" */

#include <stdio.h>

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

Podemos observar que en el texto del programa aparece una línea precedida del símbolo #. Con este símbolo comienzan lo que se llaman *directivas para el compilador*. En concreto con la directiva `#include` se indica al compilador que utilice el módulo de librería `stdio` (cabecera `<stdio.h>`) para las operaciones de escritura que se realizarán en el programa. De hecho la directiva `#include` será la única que se usará en **C±**.

El cuerpo del programa contiene las sentencias ejecutables correspondientes a las acciones a realizar, escritas entre los símbolos { de comienzo y } final. Cada sentencia del programa termina con un punto y coma (`;`).

Es conveniente recordar que todos los programas en **C±** se deben guardar en un fichero con el nombre del programa y la extensión `.cpp`. Así, el nombre del fichero fuente de este programa deberá ser: `hola.cpp`.

Una vez compilado y ejecutado este programa de ejemplo produce, como es de esperar, el siguiente resultado:

```
Hola
```

En el anterior ejemplo de programa aparece también un nuevo elemento no mencionado hasta el momento, y que se explica a continuación.

### 2.8.1 Uso de comentarios

El código de un programa en un lenguaje de programación tal como C, C++ o **C±** puede no ser suficiente, en muchos casos, para comprender el sentido del programa. Casi siempre es conveniente alguna aclaración adicional que explique el significado exacto de los elementos usados para desarrollar el programa. Estas aclaraciones facilitan la labor de una posible modificación posterior del programa por nosotros mismos u otros programadores.

Todos los lenguajes permiten incluir dentro del texto del programa *comentarios* que faciliten su comprensión. Estos comentarios sirven sólo como documentación del programa fuente, y son ignorados por el compilador, en el sentido de que no pasan a formar parte del código objeto al que se traducirá el programa.

En **C±** los comentarios se incluyen dentro de los símbolos `/*` y `*/` . Por ejemplo:

```
/* ¡Ojo!. Esto es un comentario */
```

### 2.8.2 Descripción formal de la estructura de un programa

La descripción formal de la estructura (simplificada) de un programa es la siguiente:

*Programa* **::=** **{** *Include* **}** `int main()` *Bloque*

*Include* **::=** `#include` <*Nombre_módulo*.h>

Cada directiva debe ocupar una línea del programa ella sola. En los lenguajes C y C++ hay una gran variedad de directivas, pero en **C±** se utilizará casi exclusivamente la directiva **`#include`**, que sirve para indicar que el programa utilizará un determinado módulo de librería. El parámetro *Nombre_módulo* corresponde en realidad al nombre del fichero de cabecera (*header*) del módulo.

El resto del código del programa se podría repartir en líneas de código como se desee, aunque en el *Manual de Estilo* de **C±** se exigirá un estilo de presentación uniforme, tal como se irá indicando a medida que se introduzcan los elementos del lenguaje.

La estructura de un bloque solamente puede indicarse de momento de forma simplificada, ya que sólo se han visto los elementos mínimos necesarios para escribir un programa. Por ahora diremos que un bloque puede contener una secuencia de sentencias:

*Bloque* **::=** { *Parte_ejecutiva* }

*Parte_ejecutiva* **::=** **{** *Sentencia* **}**

La única sentencia que se ha descrito hasta el momento es la orden de escritura `printf`. Más adelante se irán introduciendo nuevas sentencias de **C±**.

## 2.9 Ejemplos de programas

En este apartado se muestran programas completos que pueden ser compilados y ejecutados de manera directa e inmediata. Aunque estos ejemplos son de una gran sencillez, permiten ilustrar los conceptos introducidos en este tema.

### 2.9.1 Escribir una fecha

En este ejemplo se escribe una fecha. El día y el año se escriben como valores numéricos, y el mes como un texto.

```
/** Programa: EscribirFecha */
/* Escribe la fecha del descubrimiento de América */

#include <stdio.h>

int main() {
  printf( "%2d", 12 );
  printf( " de Octubre de" );
  printf( "%5d\n", 1492 );
}
```

La ejecución del programa produce el siguiente resultado (usando el símbolo '·' para representar el espacio en blanco):

```
12·de·Octubre·de·1492
```

Obsérvese cómo se han incluido los espacios en blanco en las sentencias de escritura.

### 2.9.2 Suma de números consecutivos

Con este programa se trata de obtener la suma de una serie de números consecutivos, desde uno inicial a otro final. Para su obtención utilizaremos la fórmula de la suma de una progresión aritmética. Si $n$ es el número de términos, $a_1$ el primer término, y $a_n$ el último, la fórmula general de la suma es:

$$\sum_{i=1}^{n} a_i = n \times \frac{a_1 + a_n}{2}$$

Y para una serie de números enteros consecutivos se cumple que:

$$n = a_n - a_1 + 1$$

El listado del programa es el siguiente:

```
/** Programa: SumarNumeros */
/* Este programa calcula e imprime la suma
   de los números correlativos desde 4 hasta 45

   El algoritmo empleado es el utilizado para
   calcular la suma de una progresión aritmética:
   Suma = (Final - Inicial + 1) * (Inicial + Final) / 2
*/
#include <stdio.h>

int main() {
  printf( "La suma de los números desde 4 hasta 45\n" );
  printf( "es igual a: ");
  printf( "%5d\n",(45 - 4 + 1) * (45 + 4) / 2 );
}
```

La ejecución del programa produce el siguiente resultado:

```
La·suma·de·los·números·desde·4·hasta·45
es·igual·a:··1029
```

Es interesante analizar la manera en que se evalúa la expresión aritmética, dado que / trunca el resultado. Si se hubiera escrito la expresión de la forma que sugiere la fórmula de la suma de la progresión aritmética, tal como se ha presentado anteriormente:

```
(45 - 4 + 1) * ( (45 + 4) / 2 )
```

se obtendría un resultado erróneo igual a 1008, al producirse el truncamiento de la división por 2 antes de multiplicar.

### 2.9.3 Área y volumen de un cilindro

En este tercer programa sencillo se obtienen el área y el volumen de un cilindro a partir de su radio $R$ y su altura $A$:

$$\begin{aligned} área &= 2\pi R^2 + 2\pi RA = 2\pi R(R+A) \\ volumen &= \pi R^2 A \end{aligned}$$

El listado del programa es el siguiente:

```
/** Programa: Cilindro */
/* Cálculo del área y el volumen de un cilindro */

#include <stdio.h>

int main() {
  printf( "%s\n", "Dado un cilindro de dimensiones:" );
  printf( "%s\n", "radio = 1,5 y altura = 5,6" );
  printf( "%s", "su area es igual a: " );
  printf( "%g\n", 2.0*3.141592*1.5*(1.5+5.6) );
  printf( "%s", "y su volumen es igual a:" );
  printf( "%20.8f\n", 3.141592*1.5*1.5*5.6 );
}
```

La ejecución del programa produce el siguiente resultado:

```
Dado un cilindro de dimensiones:
radio = 1,5 y altura = 5,6
su área es igual a: 66.9159
y su volumen es igual a:       39.58405920
```

En este ejemplo se ha renunciado a marcar expresamente los espacios en blanco impresos por el programa, para facilitar la lectura de los resultados.

# Tema 3

# Constantes y Variables

Este tema complementa el anterior con nuevos elementos que permiten construir programas algo más realistas.

Primeramente se indica el interés que tiene dar nombre a las constantes que se manejan en los programas. A continuación se introduce el concepto de variable. Este concepto es muy importante y una gran parte del tema está organizada en torno a él. De hecho la existencia de variables constituye el elemento diferenciador del paradigma de la programación imperativa respecto a los demás.

El tema se completa con la presentación de varias sentencias de lectura simple. Estas sentencias permiten construir programas en los que se pueden introducir valores para ser usados durante su ejecución.

## 3.1 Identificadores

La manera de hacer referencia a los diferentes elementos que intervienen en un programa es darles un nombre particular a cada uno. En programación se llaman *identificadores* a los nombres usados para identificar cada elemento del programa.

En el tema anterior ya se han utilizado identificadores para dar nombre a elementos tales como los tipos de datos, las operaciones de escritura, etc., que se suministran ya con un significado determinado. Ahora se inventarán nombres para designar variables y constantes en el programa.

En **C±** los identificadores son una palabra formada con caracteres alfabéticos o numéricos seguidos, sin espacios en blanco ni signos de puntuación inter-

calados, y que debe comenzar por una letra. Pueden usarse las 52 letras mayúsculas y minúsculas del alfabeto inglés, el guión bajo (_), y los dígitos decimales del 0 al 9.

Ejemplos de identificadores válidos son los siguientes:

```
Indice   diaDelMes   Nombre_Apellido
j5       Eje_3       IdentificadorMuyyyyyyLargo
```

No serían válidos los siguientes identificadores:

| | |
|---|---|
| `3_Eje` | No puede comenzar por un dígito |
| `#50$` | No se pueden usar los caracteres: # y $ |
| `dia Del Mes` | No se pueden intercalar blancos |
| `Año` | No se puede utilizar la letra ñ |

Es importante resaltar que las letras ñ o Ñ o las vocales acentuadas no están incluidas en el alfabeto de la mayoría de los lenguajes de programación, y por supuesto tampoco en **C±**, por lo que no pueden formar parte de ningún identificador. Sin embargo, estas letras sí pueden ser utilizadas como valores de tipo `char` o formando parte de una cadena.

**C±** distingue las letras mayúsculas de las minúsculas. Así, son identificadores distintos los siguientes:

```
DiaSemana   Diasemana   diaSemana   DIASEMANA
```

En general, se debe tener cuidado al utilizar identificadores que difieran en pocas letras y en particular que difieran sólo en el uso de las letras mayúsculas o minúsculas para distinguir identificadores diferentes. En estos casos, es difícil distinguir entre un cambio de alguna letra por error y una utilización correcta de identificadores distintos.

Las reglas exactas para la formación de identificadores en **C±** son las siguientes:

*Identificador* **::=** *Letra* **{** *Letra* **|** *Guión* **|** *Dígito* **}**

*Letra* **::=**
A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z

*Guión* **::=** _

*Dígito* **::=** 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A la hora de inventar nombres conviene seguir unas reglas de estilo uniformes que faciliten la lectura del programa. En el *Manual de Estilo* de **C±** se sugieren, entre otras, las siguientes:

- Por defecto, escribir todo en minúsculas:

  `indice nombre apellidos area`

- Escribir en mayúsculas o empezando por mayúsculas los nombres de constantes que sean globales o que sean parámetros generales del programa:

  `Pi NULO MAXIMO`

- Usar guiones o mayúsculas intermedias para los nombres compuestos:

  `diaDelMes ANCHO_MAXIMO dia_del_mes`

## 3.2 El vocabulario de C±

Además de los identificadores usados para nombrar diferentes elementos, en un programa podemos encontrar otras palabras que siguen las mismas reglas de formación que los identificadores pero que realmente no son nombres. Se trata de las *palabras clave*, que sirven para delimitar determinadas construcciones del lenguaje de programación.

Las palabras clave son elementos fijos del lenguaje. También son elementos fijos del lenguaje los nombres de los tipos fundamentales y los de algunas funciones especiales incorporadas en el propio lenguaje. Este conjunto de elementos fijos se denominan *palabras reservadas*, y no pueden ser redefinidas por el programador para utilizarlas con otros fines. Las palabras reservadas en el subconjunto **C±** son las siguientes:

```
bool    break  case  catch    char     const   default
delete  do     else  enum     extern   false   float
for     if     int   new      private  return  struct
switch  true   try   typedef  union    void    while
```

Aunque en **C±** sólo usaremos las palabras reservadas que se han indicado, para poder compilar los programas con compiladores de C++ deberemos considerar también como palabras reservadas todas las del lenguaje completo, y no sólo el subconjunto empleado en este libro. La lista completa de palabras reservadas es:

```
and       and_eq    asm       auto      bitand    bitor
bool      break     case      catch     char      class
compl     const     const_cast          continue  default
delete    do        double    dynamic_cast        else
enum      explicit  extern    false     float     for
friend    goto      if        inline    int       long
mutable   namespace new       not       not_eq    operator
or        or_eq     private   protected public    register
reinterpret_cast    return    short     signed    sizeof
static    static_cast         struct    switch    template
this      throw     true      try       typedef   typeid
typename  union     unsigned  using     virtual   void
volatile  wchar_t   while     xor       xor_eq
```

Además de las palabras de la lista anterior hay algunos identificadores que sin estar reservados en C++ tienen un significado preciso en cada programa en el que aparezcan. Por lo tanto tampoco deberían ser redefinidos por el programador para otros fines. Por ejemplo:

```
main  NULL  std  string  ...
```

En **C±** se considerarán también estos identificadores como palabras reservadas, y por lo tanto será un error redefinirlos.

## 3.3 Constantes

En primer lugar se introduce el concepto de constante y a continuación se explica cómo se manejan las constantes en **C±**.

### 3.3.1 Concepto de constante

Una *constante* es un valor fijo que se utiliza en un programa. El valor debe ser siempre el mismo para cualquier ejecución del programa, es decir, el valor no puede cambiar de una ejecución a otra. Son ejemplos de constantes el número de meses del año, el número de días de una semana, las constantes matemáticas tales como el número $\pi$, los factores de conversión de unidades físicas de medida, etc.

Hasta el momento se ha visto la manera de representar valores constantes en un programa escribiéndolos explícitamente, en forma de *constantes literales*, como se las denomina en programación. Otra posibilidad es definir estos valores como *constantes simbólicas* o *constantes con nombre*.

### 3.3.2 Declaración de constantes con nombre

La declaración de un valor constante con nombre consiste en asociar un identificador a dicho valor constante. La declaración de la constante especifica su nombre y tipo y el valor asociado. Si queremos declarar el valor de la constante $\pi$ asociado al nombre **Pi** escribiremos:

```
const float Pi = 3.14159265;
```

La declaración se inicia con la palabra clave **const**, y a continuación se escriben el tipo y nombre simbólico de la constante, seguidos del signo igual y el valor asociado.

Las constantes con nombre han de ser declaradas en el programa antes de ser utilizadas. Una vez definida la constante se puede utilizar su nombre exactamente igual que si fuera su valor explícito. Por ejemplo, serán equivalentes la dos expresiones siguientes:

`2 * 3.14159265 * R`   y   `2 * Pi * R`

Algunos ejemplos de declaración de constantes son los siguientes:

```
const int   largo = 34;
const int   ancho = 78;
const char  dospuntos = ':';
const float NumeroE = 2.718281828459;
const char Pregunta[] = "¿Código postal?";
const float radio = 1.5;
```

Como puede verse, el tipo de valor de una constante se declara explícitamente, y no viene forzado por el valor constante literal que se le asocia. En el caso de la constante **Pregunta** su tipo es cadena de caracteres y la forma en que se declara es utilizando los símbolos **[]** a continuación del nombre de la constante. La siguientes constantes tienen tipos diferentes, aunque se use el mismo valor literal para ambas:

```
const int   minimaTemperatura = -50;
const float temperaturaMinima = -50;
```

Una posibilidad interesante es poder declarar el valor de una constante en forma de expresión. En **C±** sólo se permite hacer esto si la expresión puede ser evaluada por el compilador en el momento de traducir el programa fuente a programa objeto. Para ello es necesario que todos los operandos que intervengan en la expresión sean valores constantes, y que las operaciones entre ellos sean operadores fijos del lenguaje o funciones predefinidas (que tienen

identificadores predefinidos). En este caso la expresión se denomina *expresión constante*. Los operandos constantes pueden ser valores explícitos o constantes con nombre declaradas en algún punto anterior del programa.

Por ejemplo, tras las declaraciones de los ejemplos anteriores se podría añadir:

```
const float diametro = 2*radio;
const int   constanteRara= (23 * 5) / ((7 - 4) % 2));
const int   area = largo * ancho;
const int   perimetro = 2*(largo + ancho);
```

Todas estas constantes con nombre se pueden utilizar exactamente igual que el valor literal que representan. Por ejemplo, el resultado de las siguientes operaciones de escritura sería:

| Operación de escritura | Resultado |
|---|---|
| `printf( "%s ", Pregunta );` | `¿Código·postal?·` |
| `printf( "%9.2d#%5d", radio, area );` | `·····1.50#·2652` |
| `printf( "%c", dospuntos );` | `:` |

En la primera de estas operaciones de escritura se hace uso del código de formato `%s` para la escritura de una cadena de caracteres con el procedimiento `printf`.

Las reglas precisas que han de seguir las declaraciones de constantes con nombre son las siguientes:

*Declaración_de_constante* **::=**
  `const` *Tipo Nombre* = *Expresión_constante* ;

*Tipo* **::=** *Identificador*

*Nombre* **::=** *Identificador*

Una expresión constante incluye, como caso particular, un único término que sea un valor explícito.

## 3.4 Variables

En este apartado se introduce el concepto de variable y a continuación se explica cómo se declaran y utilizan las variables en **C±**.

### 3.4.1 Concepto de variable

El concepto de *variable* en programación imperativa es diferente del concepto de variable algebraica. Cuando escribimos expresiones algebraicas usamos

variables para representar un valor indefinido, pero fijo. Por ejemplo, cuando escribimos la igualdad:

$$(a+b)^2 = a^2 + 2ab + b^2$$

queremos indicar que dicha igualdad se cumple asociando $a$ y $b$ con valores cualesquiera, pero siempre los mismos. No tiene sentido pretender que $a$ tenga un valor en una parte de la expresión y otro diferente en otra parte.

Algo similar ocurre cuando escribimos fórmulas matemáticas o leyes físicas, que ligan diversas magnitudes. Por ejemplo, la fórmula del volumen ($V$) de un cilindro en función de su radio ($R$) y su altura ($A$) es:

$$V = \pi R^2 A$$

Una vez asociadas las variables $R$ y $A$ con el radio y altura de un cilindro en particular, la variable $V$ debe tomar necesariamente el valor del volumen de dicho cilindro.

Insistiendo en esta idea, consideremos un sistema de ecuaciones, tal como:

$$3x + 5y = 19$$
$$7x - 5y = 11$$

Las variables $x$ e $y$, denominadas incógnitas, representan los valores, inicialmente desconocidos, que satisfacen dichas ecuaciones, y en concreto:

$$x = 3 \quad y = 2$$

Con todo ello insistimos en que las variables algebraicas representan valores simbólicos, bien valores cualesquiera, o bien valores desconocidos, pero de manera que una vez asociada la variable a un valor determinado dicho valor no debe cambiarse.

En programación el concepto de variable es diferente, y está directamente asociado a la memoria del computador. Esta memoria permite almacenar información para ser usada posteriormente. La función de la memoria es mantener dicho valor todo el tiempo que sea necesario para usarlo tantas veces como se necesite.

Los valores almacenados en la memoria pueden ser modificados cuantas veces se desee. Al almacenar un valor en un elemento determinado de la memoria, dicho valor se mantiene de ahí en adelante, pero sólo hasta que se almacene en dicho elemento un nuevo valor diferente.

Las variables en los lenguajes de programación imperativos son el concepto abstracto equivalente a la memoria física de la máquina. Una variable representa un valor almacenado que se puede conservar indefinidamente para ser

usado tantas veces como se desee. El valor de una variable se puede modificar en cualquier momento, y será el nuevo valor el que estará almacenado en ella a partir de entonces.

Las variables de un programa se designan mediante nombres o identificadores. El identificador de una variable representa el valor almacenado en dicha variable. El programador puede elegir los nombres que considere más apropiados para las variables que utilice en su programa.

### 3.4.2 Declaración de variables

Cada variable en un programa en **C±** debe tener asociado un tipo de valor determinado. Esto quiere decir que si una variable tiene asociado el tipo `int`, por ejemplo, sólo podrá almacenar valores de este tipo, pero no valores de tipo `float` u otro diferente.

Las variables han de ser declaradas en el programa antes de ser utilizadas. La declaración simple de una variable especifica su nombre y el tipo de valor asociado. Por ejemplo, para usar una variable que almacene la edad de una persona como un número entero de años, podríamos declarar:

```
int edad;
```

Podemos observar que la declaración consiste simplemente en escribir el tipo y el nombre de la variable. La declaración termina con punto y coma (`;`).

Si varias variables tienen el mismo tipo, se pueden declarar todas conjuntamente, escribiendo sus nombres seguidos, separados por el carácter coma (`,`) detrás del tipo común a todas ellas. Por ejemplo:

```
int dia, mes, anno;
```

La descripción BNF (simplificada) de una declaración de variables es la siguiente:

*Declaración_de_variable* **::=** *Tipo Nombre* **{** , *Nombre* **}** ;

*Tipo* **::=** *Identificador*

*Nombre* **::=** *Identificador*

Por el momento las posibilidades de especificación de tipos se limitan a los tipos predefinidos presentados hasta ahora; más adelante se indicará la manera de definir nuevos tipos.

### 3.4.3 Uso de variables. Inicialización

El valor almacenado en una variable puede utilizarse usando la variable como operando en una expresión aritmética. El tipo declarado para cada una de las variables determina las operaciones que posteriormente se podrán realizar con ella, de la misma forma que sucedía para los valores literales según se explicó en el tema anterior.

Por ejemplo, si declaramos las variables:

```
int   base, altura;
int   saldo, meses, dias;
float volumen, area, gastos;
char  modelo, codigo;
```

podremos escribir expresiones tales como:

```
base * altura
dias + int( codigo )
volumen / area
```

pero sería inapropiado escribir

```
area / base
saldo + gastos
base + modelo
```

porque combinan operandos de tipos diferentes.

Si se usan correctamente los tipos de operandos para cada operación, en una misma expresión pueden intervenir operandos de diferentes clases. Por ejemplo, pueden usarse a la vez variables, constantes con nombre y valores numéricos constantes. Con las declaraciones de constantes de la sección anterior y las variables de los últimos ejemplos se podría escribir:

```
Pi * volumen * 5.7
base * altura / constanteRara
float( saldo ) + NumeroE
int( area ) / ancho % 5
```

Estos ejemplos de expresiones son formalmente correctos, de acuerdo con las reglas de **C±**, pero no se pretende que tengan ningún sentido intuitivo que se ajuste al significado de los nombres empleados.

Con independencia de que las operaciones con variables sean consistentes con los tipos de valores que almacenan, para que sea válido evaluar una expresión aritmética en la que intervengan variables es necesario que todas ellas tengan

un valor definido. Tal como se ha dicho anteriormente, los nombres de las variables representan los valores almacenados en ellas. Si en una variable no se ha almacenado todavía ningún valor, su uso conduce a un resultado imprevisible.

Para usar una variable de manera correcta es necesario inicializarla antes de usar su valor en ningún cálculo. *Inicializar* una variable es simplemente darle un valor determinado por primera vez.

Para evitar confusiones, en el *Manual de Estilo* de **C±** sólo está permitido especificar el valor inicial de una variable en una declaración individual para esa variable. Por tanto, no está permitido realizar inicializaciones de ninguna variable cuando se declaran como una lista de variables del mismo tipo. Para inicializar la variable, su nombre irá seguido del signo igual (=) y a continuación su valor inicial, de forma similar a una declaración de constante:

```
float gastos = 0.0;
char modelo = '?';
```

Como no es obligatorio dar valor inicial a todas las variables, unas pueden llevar valor inicial y otras no:

```
int   base=0;
int   altura=0;
float volumen, area, gastos;
char  modelo='?';
char  codigo;
```

La descripción BNF (corregida) de la declaración de variables, con valor inicial opcional, es la siguiente:

*Declaración_de_variable* **::=** *Variable_simple* **|** *Lista_de_variables* ;

*Variable_simple* **::=** *Tipo Nombre* **[** = *Expresión* **]**

*Lista_de_variables* **::=** *Tipo Nombre* **{** , *Nombre* **}**

*Tipo* **::=** *Identificador*

*Nombre* **::=** *Identificador*

Si no se especifica el valor inicial al declarar la variable, entonces deberá ser inicializada en el momento adecuado asignándole valor de alguna manera durante la ejecución del programa, antes de usar el valor almacenado para operar con él.

## 3.5 Sentencia de asignación

Una forma de conseguir que una variable guarde un determinado valor es mediante una *sentencia de asignación*. Esta sentencia es característica de la programación imperativa, y permite inicializar una variable o modificar el valor que tenía hasta el momento. Mediante asignaciones podremos dar valores iniciales a las variables, o guardar en ellas los resultados intermedios o finales de cualquier programa.

La estructura de una sentencia de asignación es la siguiente:

*Asignación* **::=** *Variable* = *Expresión* ;

*Variable* **::=** *Identificador*

Por el momento la única manera que se ha presentado de hacer referencia a una variable es usando su nombre. Más adelante se indicarán otras formas de hacer referencia a variables, especialmente en el caso de que la variable designada sea parte de una estructura de datos.

El signo igual (=) es el operador de asignación. Este operador indica que el resultado de la expresión a su derecha debe ser asignado a la variable cuyo identificador está a su izquierda. Por ejemplo, la secuencia de asignaciones siguiente:

```
base = 18;
area = 56.89;
codigo = 'z';
```

sustituye los valores que tuvieran las variables **base**, **area** y **codigo** hasta ese momento por los nuevos valores 18, 56,89 y la letra "z", respectivamente.

Si intervienen variables en la expresión a la derecha de una sentencia de asignación, se usará el valor que tenga la variable en ese momento. Por ejemplo, la siguiente secuencia de asignaciones:

```
meses = 2;
dias  = meses;
meses = 7;
saldo = meses;
```

dejará finalmente almacenados en las variables **meses**, **dias** y **saldo** los valores 7, 2 y 7, respectivamente.

Un caso especial, que requiere cierta atención, es aquél en que a una variable se le asigna el valor de una expresión de la que forma parte la propia variable. Por ejemplo:

```
dias = dias + 30;
```

Esta asignación recuerda a una ecuación, pero su significado es completamente diferente. Esta sentencia es una orden de evaluar primero la expresión a la derecha usando el valor que tenía la variable hasta ese momento, y luego modificar el valor de la variable almacenando en ella el resultado de la expresión. En este ejemplo, si la variable `dias` tenía el valor 16, esta sentencia almacenará en ella el nuevo valor 46. En general, al ejecutar esta sentencia de asignación, el valor de la variable `dias` se incrementará en 30 unidades.

### 3.5.1 Sentencias de autoincremento y autodecremento

Resulta bastante frecuente la necesidad de incrementar en uno el valor de una variable. Para ello, empleando la sentencia de asignación se tiene que escribir:

```
variable = variable + 1;
```

Para simplificar y aumentar la claridad del programa, en **C±** se dispone de una sentencia especial de *autoincremento* que utiliza el símbolo ++. Así, en lugar de la sentencia de asignación anterior se suele utilizar habitualmente la sentencia de autoincremento de la siguiente manera:

```
variable++;
```

De forma semejante cuando se necesita decrementar una variable en una unidad, **C±** dispone de la sentencia especial de *autodecremento* que utiliza el símbolo -- y que se escribe:

```
variable--;
```

Esta sentencia es equivalente a la sentencia de asignación:

```
variable = variable - 1;
```

### 3.5.2 Compatibilidad de tipos

Una sentencia de asignación puede resultar confusa si el tipo de la variable y el del resultado de la expresión son diferentes, de manera similar a lo que ocurre con una operación aritmética entre operandos de tipos diferentes. El lenguaje **C±** no es *fuertemente tipado* y permite la ambigüedad que supone la asignación de un valor de un tipo a una variable de otro tipo. Para salvar la ambigüedad, **C±** utiliza el convenio de C/C++ de convertir previamente de manera automática el valor a asignar al tipo del valor de la variable. De todas

maneras, para evitar confusiones, en el *Manual de Estilo* se establece que para la realización de programas en **C±** es obligatorio que se realice siempre una conversión explícita de tipos en estos casos. Ejemplo:

```
int   saldo;
float gastos;
...
saldo = int(gastos);
```

Para finalizar este apartado se presenta una declaración de variables y luego la traza de una secuencia de sentencias de asignación. En el lado izquierdo se muestran los valores que guardan las variables inmediatamente antes de la ejecución de cada sentencia. Nótese que las variables tienen un valor indefinido hasta que son inicializadas asignándoles valor por primera vez.

Declaración:

```
int posi, dato;
float ejeX, ejeY;
```

Traza de ejecución:

| ejeX | ejeY | dato | posi | Secuencia de sentencias |
|---|---|---|---|---|
| ? | ? | ? | ? | `ejeX = 34.89;` |
| 34.89 | ? | ? | ? | `dato = 67;` |
| 34.89 | ? | 67 | ? | `posi = int(ejeX) + dato;` |
| 34.89 | ? | 67 | 101 | `dato = int(ejeY) + posi;` |
| 34.89 | ? | ? | 101 | `ejeY = ejeX + float(posi);` |
| 34.89 | 135.89 | ? | 101 | `dato = posi / 5;` |
| 34.89 | 135.89 | 20 | 101 | `posi = posi % 5;` |
| 34.89 | 135.89 | 20 | 1 | `posi = posi * dato;` |
| 34.89 | 135.89 | 20 | 20 | `ejeY = ejeY/ejeX;` |
| 34.89 | 3.8948 | 20 | 20 | `ejeX = ejeX/2.0;` |
| 17.44 | 3.8948 | 20 | 20 | `posi = int(ejeY) - posi;` |
| 17.44 | 3.8948 | 20 | -17 | ... |

Los datos representados con interrogación (?) significan valores imprevisibles, debidos a variables no inicializadas cuyo contenido no puede conocerse con seguridad. Obsérvese que la variable `dato`, después de tomar un valor definido (67), vuelve a tomar un valor arbitrario al asignársele el resultado de una expresión en la que participa la variable `ejeY` que todavía estaba sin inicializar.

## 3.6 Operaciones de lectura simple

Los programas presentados hasta el momento producen siempre el mismo resultado cada vez que se mandan ejecutar. Por ejemplo, con los elementos introducidos hasta ahora se podrían escribir programas para calcular la suma de los números del 4 al 45 o imprimir la nómina de Febrero del Sr. González. Programas que produzcan resultados fijos son poco habituales en la práctica. Bastaría ejecutarlos una sola vez, guardar los resultados y reproducirlos tantas veces como se necesiten, sin tener que repetir la ejecución del programa.

Los programas habituales suelen resolver problemas genéricos. Por ejemplo, obtener la suma de $N$ números desde uno inicial a otro final, o calcular cada mes la nómina de cada uno de los empleados de una empresa. Un programa que produzca cada vez resultados diferentes deberá operar en cada caso a partir de unos datos distintos, y dichos datos no pueden ser, por tanto, valores constantes que formen parte del programa.

Para resolver cada vez el problema concreto que se plantea, el programa debe leer como datos de entrada los valores concretos a partir de los cuales hay que obtener el resultado. Por ejemplo, los valores inicial y final de la serie de números a sumar, o el nombre del empleado cuya nómina se quiere calcular y el mes al que corresponde. Por consiguiente, las operaciones de lectura son fundamentales dentro de cualquier modelo de programación.

En **C±**, los datos leídos han de ser guardados inmediatamente en variables del programa. Por tanto, otra manera de asignar un valor a una variable es almacenar en ella un valor introducido desde el exterior del computador mediante el teclado u otro dispositivo de entrada de datos.

Las *operaciones de lectura*, al igual que las de escritura, pueden presentar grandes diferencias dependiendo del dispositivo utilizado. Pero, también en este caso, existe en **C±** un conjunto de procedimientos generales para la lectura de datos, que son invocados siempre de la misma manera, con independencia del dispositivo de entrada utilizado. Por defecto, el dispositivo de entrada suele estar asociado al teclado del terminal por el que se accede al computador. Estos procedimientos están incluidos también en el *módulo de librería* `stdio`.

Los datos a leer se suministran externamente en forma de texto, es decir, como una serie de caracteres seguidos (o pulsaciones de teclas) que pueden incluir saltos de línea de vez en cuando. En el teclado, el salto de línea corresponde a la tecla marcada “INTRO” o “ENTRAR” (en teclados ingleses, “RETURN” o “ENTER”).

### 3.6.1 El procedimiento scanf

El procedimiento `scanf` pertenece al módulo de librería `stdio`. Para leer datos de entrada y almacenarlos en determinadas variables se escribirá:

`scanf(` *cadena-con-formatos*`, &`*variable1*`, &`*variable2*`, ... &`*variableN* `);`

Es importante observar que los nombres de las variables a leer van precedidos del carácter *ampersand* (&).

A continuación se presenta un ejemplo de uso de este procedimiento de lectura.

| | |
|---|---|
| Declaración de variables | `int mes, dia; float saldo;` |
| Datos de entrada | `123 4.5 6` |
| Orden de lectura | `scanf( "%d %f %d", &mes, &saldo, &dia );` |
| Resultado | `mes = 123; saldo = 4.5; dia = 6;` |

La cadena de caracteres con los formatos sigue las mismas reglas que la del procedimiento `printf` del tema anterior. Debe contener un formato de conversión (%$x$) por cada variable a leer. Al ejecutarse el procedimiento `scanf` se van tomando caracteres del dispositivo de entrada, se ponen en correspondencia con los formatos indicados, y se extraen los valores correspondientes a cada formato de conversión para asignarlos a cada variable de la lista, respectivamente.

Al igual que en el procedimiento `printf`, la cadena con los formatos puede incluir también otros fragmentos de texto entremezclados con los formatos de conversión. La ejecución de `scanf` consiste en analizar uno a uno los elementos de la cadena con formatos, y actuar en consecuencia, avanzando en el texto de entrada:

1. Un formato numérico (`%d`, `%f`, `%g`, ...) hace que se salten los siguientes caracteres de espacio en blanco en la entrada, si los hay. A continuación se leen los caracteres no blancos que formen una representación válida de un valor numérico del tipo correspondiente al formato, y el valor numérico obtenido se asigna al siguiente argumento de la lista de variables a leer.
2. Un formato `%c` hace que se lea exactamente el siguiente carácter de la entrada, sea o no espacio en blanco, y se asigne a la siguiente variable a leer.
3. Un carácter de espacio en blanco en la cadena con formatos hace que se salten los siguientes caracteres de espacio en blanco en el texto de entrada, si los hay.
4. Un carácter no blanco en la cadena con formatos hace que se lea (y se salte) el siguiente carácter de la entrada, que debe coincidir exactamente con el carácter del formato.

Si alguna de las acciones anteriores no puede realizarse, porque el texto de entrada no contiene los caracteres apropiados, la ejecución de `scanf` se interrumpe en ese momento, y no se lee más texto de entrada ni se asignan valores a las variables que falten por leer.

Los formatos de conversión numéricos pueden incluir la especificación del tamaño del dato, de forma similar a como se dijo para `printf`. Pero a diferencia de los formatos para `printf`, en que ese tamaño era el mínimo número de caracteres a escribir, en `scanf` significa el tamaño máximo del dato de entrada a leer. Por ejemplo:

| Datos | Formato | Dato leído | Datos restantes |
|---|---|---|---|
| 12345xx | %d | 12345 | xx |
| 12345xx | %3d | 123 | 45xx |
| 12xx345 | %3d | 12 | xx345 |

### 3.6.2 Lectura interactiva

Cuando un programa se comunica con el usuario mediante un terminal de texto se suele programar cada operación de lectura inmediatamente después de una escritura en la que se indica qué dato es el que se solicita en cada momento. Por ejemplo:

```
float saldo
...
printf( "¿Cantidad Pendiente? " );
scanf( "%f", &saldo );
```

Tras ejecutar la escritura del texto de petición, en la pantalla se verá:

```
¿Cantidad Pendiente? ▬
```

En ese momento se inicia la ejecución del procedimiento de lectura, El símbolo (▬) se utiliza aquí para indicar la posición del cursor en la pantalla, en espera de la entrada del dato solicitado.

Cuando se introduce información por el teclado es ventajoso que se pueda corregir el texto que se va tecleando en caso de cometer errores. Para ello la lectura de teclado se suele hacer por líneas completas. Mientras se teclea el texto de una línea se pueden hacer correcciones, y al pulsar la tecla de fin de línea el texto se acepta y se procesa como dato de entrada. El texto aparece en la pantalla al mismo tiempo que se va tecleando o corrigiendo. Todo esto es automático, y se hace directamente al invocar el procedimiento de lectura de **C±**.

En el ejemplo anterior, se tecleará el dato acabando con la tecla de fin de línea, que representaremos como Intro:

```
¿Cantidad Pendiente? -45768 Intro
▬
```

Al pulsar Intro el cursor pasará a la siguiente línea, se procesará el dato de entrada, y a la variable **saldo** se le asignará el nuevo valor -45768.

Si se quiere introducir más de un valor con la misma pregunta y en la misma línea se puede utilizar como separador de datos el espacio en blanco. Por ejemplo:

```
float saldo, gastos
...
printf( "¿Cantidad pendiente y Gastos? " );
scanf( "%f%f", &saldo, &gastos );
```

En la pantalla aparecerá el mensaje de petición, y luego se teclearán los datos separados por uno o varios blancos y acabados con un Intro:

```
¿Cantidad pendiente y Gastos? -45768 10456.5 Intro
▬
```

Al pulsar Intro queda el cursor en la línea siguiente, la variable **saldo** toma el valor -45768 y la variable **gastos** el valor 10456.5. En este caso el espacio en blanco intermedio entre los datos hace que termine la lectura del primer valor y se pase a leer el segundo, saltando automáticamente dicho espacio y los siguientes si los hubiera.

El mismo resultado se obtendría programando cada lectura como una sentencia separada, o introduciendo los datos en varias líneas.

```
float saldo, gastos
...
printf( "¿Cantidad pendiente y Gastos? " );
scanf( "%f", &saldo );
scanf( "%f", &gastos );
```

```
¿Cantidad pendiente y Gastos? -45768 Intro
10456.5 Intro
▬
```

Cualquier combinación es válida: una sentencia única para leer dos valores o dos sentencias para leer un valor cada una, e introducir los datos en una o dos líneas, indistintamente.

## 3.7 Estructura de un programa con declaraciones

Antes de realizar nuevos programas es necesario saber dónde se sitúan dentro del programa los nuevos elementos introducidos en este tema. Las declaraciones de valores constantes y de las variables forman parte del bloque del programa. Por tanto, ahora la estructura del bloque es la siguiente:

*Bloque* **::=** { *Parte_declarativa Parte_ejecutiva* }

*Parte_declarativa* **::=** **{** *Declaración* **}**

*Parte_ejecutiva* **::=** **{** *Sentencia* **}**

*Declaración* **::=**
*Declaración_de_constante* **|** *Declaración_de_variable* **|** ...

*Sentencia* **::=** *Llamada_a_procedimiento* **|** *Asignación* **|** ...

El contenido de un bloque se organizará en dos partes. La primera de ellas contendrá todas las declaraciones de constantes, variables, etc., y la segunda incluirá las sentencias ejecutables correspondientes a las acciones a realizar. Las declaraciones pueden hacerse en el orden que se quiera, con la limitación de que cada nombre debe ser declarado antes de ser usado. Las sentencias ejecutables deben escribirse exactamente en el orden en que han de ser ejecutadas.

▮ NOTA: En C y C++ es posible mezclar las declaraciones con las sentencias ejecutables pero esto está absolutamente prohibido en **C±** con una regla obligatoria del *Manual de Estilo* y la sintaxis del lenguaje **C±** que está recogida en el apéndice A. La separación de la parte declarativa y la parte ejecutiva de un bloque está incorporada en la sintaxis de muchos lenguajes de programación (Pascal, Modula-2, Oberon, Ada, Smalltalk, etc.) y además es una regla de estilo bastante habitual en proyectos reales para evitar problemas de uso y aumentar la claridad de los programas.

## 3.8 Ejemplos de programas

A continuación se presentan varios programas con los nuevos conceptos introducidos.

### 3.8.1 Ejemplo: Conversión a horas, minutos y segundos

En este ejemplo se trata de convertir a horas, minutos y segundos una cierta cantidad de tiempo expresada en segundos. La variable **segundos** se utilizará para leer la cantidad total de segundos y posteriormente para ir guardando

los segundos restantes al descontar del total de segundos los que se hayan convertido a horas o minutos completos. El listado del programa es el siguiente:

```
/** Programa: HorasMinutosSegundos */
/* Conversión a Horas, Minutos y Segundos
   de los segundos introducidos como dato */

#include <stdio.h>

int main() {
  int horas, minutos, segundos;

  printf( "¿Segundos Totales? " );
  scanf( "%d", &segundos );
  horas = segundos / 3600;
  segundos = segundos % 3600;
  minutos = segundos / 60;
  segundos = segundos % 60;
  printf( "Equivalen a %2d horas %2d min. y %d seg.\n",
          horas, minutos, segundos );
}
```

La ejecución del programa introduciendo como dato 23459 segundos es la siguiente:

```
¿Segundos Totales? 23459
Equivalen a 6 horas, 30 min. y 59 seg.
```

### 3.8.2 Ejemplo: Área y volumen de un cilindro

En este ejemplo se desarrolla un programa para el cálculo del área y el volumen de un cilindro genérico. Es decir, las dimensiones del cilindro serán variables y se leerán como datos. Además se declara como constante el valor de $\pi$ lo que evita tener que escribirlo más de una vez en todo el programa. El programa así realizado es más general y más fácil de comprender y modificar que la versión presentada en el tema anterior. El listado del programa es el siguiente:

```
/** Programa: Cilindro2 */
/* Cálculo del área y el volumen de un cilindro */

#include <stdio.h>
```

```
int main() {
  const float PI = 3.14159265;
  float radio, altura, area, volumen;

  printf( "¿Radio del cilindro? " );
  scanf( "%f", &radio );
  printf( "¿Altura del cilindro? " );
  scanf( "%f", &altura );
  area = 2.0 * PI * radio * (radio + altura);
  volumen = PI * radio * radio * altura;
  printf( "Area: %15g\nVolumen: %15g\n", area, volumen );
}
```

La ejecución del programa introduciendo como valor del radio 12,5 y como altura 72,61 produce el siguiente resultado:

```
¿Radio del cilindro? 12.5
¿Altura del cilindro? 72.61
Area:          6684.52
Volumen:          35642.4
```

### 3.8.3 Ejemplo: Realización de un recibo

En este ejemplo se trata de confeccionar un recibo sencillo, correspondiente a la compra de unos equipos. Como datos habrá que introducir el tipo de equipo, indicado mediante un código de un carácter; la cantidad de equipos, el precio unitario y el tipo de IVA aplicado. Los subtotales y totales se darán en euros y céntimos.

El listado del programa completo es el siguiente.

```
/** Programa: Recibo */
/* Cálculo e impresión de un recibo */

#include <stdio.h>

int main() {
  int cantidad, IVA;
  char codigo;
  float precio, totalIVA, subtotal, total;

```

```
  printf( "¿Código del producto? " );
  scanf( "%c", &codigo );
  printf( "¿Cantidad? " );
  scanf( "%d", &cantidad );
  printf( "¿Precio unitario? " );
  scanf( "%f", &precio );
  printf( "¿IVA aplicable? " );
  scanf( "%d", &IVA );
  subtotal = float(cantidad) * precio;
  totalIVA = subtotal * float(IVA) / 100.0;
  total = subtotal + totalIVA;
  printf( "\n              RECIBO de COMPRA\n\n" );
  printf( "Cantidad   Concepto     Euros/unidad   Total\n" );
  printf( "%5d     Producto: %c  %12.2f%12.2f\n\n",
          cantidad, codigo, precio, subtotal );
  printf( "%28d%% IVA  %12.2f\n\n", IVA, totalIVA );
  /* se ha codificado %% para imprimir literalmente un % */
  printf( "                           TOTAL%14.2f\n", total );
}
```

A continuación se presenta un ejemplo de la ejecución del programa, mostrando los datos leídos y los resultados escritos.

```
¿Código del producto? A
¿Cantidad? 12
¿Precio unitario? 2345
¿IVA aplicable? 16

              RECIBO de COMPRA

Cantidad    Concepto     Euros/unidad    Total
   12     Producto: A       2345.00    28140.00

                          16% IVA       4502.40

                            TOTAL      32642.40
```

# Tema 4

# Metodología de Desarrollo de Programas (I)

Aquí se tratan explícitamente los primeros conceptos metodológicos relacionados con la programación en general y con la programación imperativa en particular. Se presenta el desarrollo por refinamientos sucesivos, aunque limitado al empleo de la estructura secuencial, por el momento.

Como elementos complementarios, se hace explícita la necesidad de usar un buen estilo de programación, que se concreta en sugerencias sobre el uso de una notación apropiada para la representación de los elementos del programa y la presentación de su estructura de forma clara.

## 4.1 La programación como resolución de problemas

La labor de programación puede considerarse como un caso particular de la resolución de problemas. Resolver un problema consiste esencialmente en encontrar una estrategia a seguir para conseguir la solución. En la figura 4.1 se plantea un problema sencillo.

Una estrategia se expresará como una colección de reglas o recomendaciones que, si se siguen, conducirán a la solución. Por ejemplo, para resolver el problema de obtener el área de un rectángulo del que se conocen las longitudes de sus lados (base y altura), formularemos la estrategia:

*multiplicar la base por la altura*

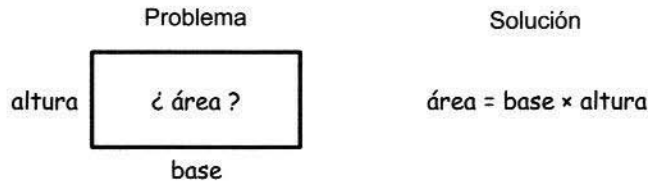



Figura 4.1 Problema: Área de un rectángulo.

Un programa en el modelo de programación imperativa se expresa como una serie de instrucciones u órdenes que gobiernan el funcionamiento de una máquina. La máquina va ejecutando dichas instrucciones en el orden preciso que se indique. La estrategia del ejemplo anterior se expresaría en **C±** como:

```
area = base * altura
```

Todo programa puede considerarse, de alguna forma, como la solución de un problema determinado, consistente en obtener una cierta información de salida a partir de unos determinados datos de entrada. La tarea de desarrollar dicho programa equivale, por tanto, a la de expresar la estrategia de resolución del problema en los términos del lenguaje de programación utilizado.

## 4.2 Descomposición en subproblemas

Algunos problemas, como el ejemplo anterior referente al área del rectángulo, pueden resolverse o programarse de forma inmediata porque la estrategia se puede expresar como una acción que se da por sabida, o como una sentencia en el lenguaje de programación elegido. Los problemas resolubles de esta manera serán, en general, problemas simples o triviales.

Cualquier problema de cierta complejidad necesitará una labor de desarrollo para expresar la solución. El método más general de resolución de problemas no triviales consiste en descomponer el problema original en subproblemas más sencillos, continuando el proceso hasta llegar a subproblemas que puedan ser resueltos de forma directa.

Lo que se busca aquí es una analogía con la labor de desarrollo de programas. Puesto que el proceso de ejecución de un programa imperativo consiste en la realización de las sucesivas acciones indicadas por las órdenes que constituyen el programa, analizaremos la resolución de problemas en que la estrategia de solución consiste en realizar acciones sucesivas.

Según esta idea, para desarrollar la estrategia de resolución, habrá que ir identificando subproblemas que se resolverán ejecutando acciones cada vez más simples. Consideremos el siguiente problema:

*Problema* : Obtener una caja de madera barnizada

Para expresar la estrategia de solución de forma imperativa comenzamos por formular la solución como una acción global que consigue el objetivo propuesto. En nuestro caso será:

0) Construir una caja de madera barnizada

En este primer nivel de formulación nos encontramos con una acción a realizar que es demasiado complicada para ejecutarla de forma inmediata. Será necesario descomponer el problema original en subproblemas más sencillos, que puedan ser resueltos mediante acciones más simples. Un primer paso de descomposición sería:

1) Obtener las piezas de madera
2) Montar la caja
3) Barnizarla

El proceso de descomposición en subproblemas debe continuar hasta que los subproblemas se puedan resolver mediante acciones consideradas directamente ejecutables por el agente que ha de proporcionar la solución. La expresión final de la solución del problema debe tener en cuenta, por tanto, qué acciones particulares se consideran realizables de forma directa. Tenemos así una analogía con la tarea de programación, que exige redactar el programa con los elementos particulares del lenguaje de programación elegido.

En nuestro ejemplo habrá que decidir si el subproblema 1) ha de considerarse resoluble mediante una acción simple o compuesta. Si podemos adquirir las piezas directamente en una tienda de "bricolaje", podremos considerar que el subproblema es resoluble con una acción simple que no necesita descomponerse. Si en la tienda sólo podemos comprar un tablero de madera sin cortar, tendremos que descomponer el problema de obtener las piezas en subproblemas más sencillos, tales como:

1.1) Obtener un tablero de madera
1.2) Dibujar sobre él la silueta de las piezas
1.3) Recortar el tablero siguiendo la silueta

De igual manera habría que proceder con los subproblemas planteados hasta este momento: 1.1), 1.2), 1.3), 2) y 3), decidiendo si son resolubles de forma inmediata o bien descomponiéndolos sucesivamente hasta llegar a acciones simples.

## 4.3 Desarrollo por refinamientos sucesivos

La aplicación de las ideas anteriores a la construcción de programas conduce a la técnica de desarrollo mediante *refinamientos sucesivos*. Esta técnica es parte de las recomendaciones de una metodología general de desarrollo de programas denominada *programación estructurada*, que se estudiará con más detalle en el tema siguiente. La técnica de refinamientos consiste en expresar inicialmente el programa a desarrollar como una acción global, que si es necesario se irá descomponiendo en acciones más sencillas hasta llegar a acciones simples que puedan ser expresadas directamente como sentencias del lenguaje de programación.

Cada paso de refinamiento consiste en descomponer cada acción compleja en otras más simples. Esta descomposición exige:

- Identificar las acciones componentes.
- Identificar la manera de combinar las acciones componentes para conseguir el efecto global.

La forma en que varias acciones se combinan en una acción compuesta constituye el *esquema* de la acción compuesta. La programación estructurada recomienda el uso de esquemas particularmente sencillos, que se indicarán más adelante. Por el momento presentamos un primer esquema que denominaremos *esquema secuencial*, que consiste en realizar una acción compuesta a base de realizar una tras otra, en secuencia, dos o más acciones componentes. Este esquema secuencial es el que se ha utilizado en el ejemplo del problema de construir la caja de madera.

### 4.3.1 Desarrollo de un esquema secuencial

La metodología de refinamientos incluye el ir desarrollando a la vez las sentencias del programa que realizan las acciones de la parte ejecutable, y la definición de las variables necesarias para almacenar la información manipulada por dichas acciones. Para desarrollar una acción compuesta según un esquema secuencial se necesitará:

(a) Identificar las acciones componentes de la secuencia. Identificar las variables necesarias para disponer de la información adecuada al comienzo de cada acción, y almacenar el resultado.
(b) Identificar el orden en que deben ejecutarse las acciones componentes.

Para ilustrar esta técnica consideraremos un caso simple, tal como el de obtener la suma de dos números enteros. Los dos números se introducirán como datos y

el programa suministrará como resultado su suma. Aunque resulte un ejemplo trivial, sirve perfectamente para ilustrar la metodología de desarrollo.

Procediendo paso a paso, describiremos de manera informal cada elemento del desarrollo, seguido de su codificación en **C±** .

(a) *Acciones componentes:*

- Cálculos: obtener la suma

```
suma = dato1 + dato2
```

- Operaciones de entrada: leer datos

```
printf( "Dar dos números: " );
scanf( "%d", &dato1 );
scanf( "%d", &dato2 );
printf( "\n" );
```

- Operaciones de salida: imprimir resultado

```
printf( "La suma es%10d\n", suma) ;
```

*Variables necesarias* : datos y resultado

```
int dato1, dato2, suma;
```

(b) *Orden de ejecución:*

1) Leer los datos
2) Calcular la suma
3) Imprimir el resultado

Si nos limitamos a describir la manera en que la acción global del programa se va descomponiendo en acciones cada vez más sencillas, podemos usar la siguiente notación de refinamiento:

*Acción compuesta* $\longrightarrow$
    *Acción 1*
    *Acción 2*
    ... etc. ...

En esta notación se utiliza una flecha ($\longrightarrow$) para indicar que una acción complicada se descompone o refina en otras más sencillas. Aplicando esta notación al ejemplo anterior, hasta llegar a sentencias de **C±**, tendremos:

*Obtener la suma de dos números* $\longrightarrow$
    *Leer los datos*
    *Calcular la suma*
    *Imprimir el resultado*

*Leer los datos* ⟶

```
printf( "Dar dos números: " );
scanf( "%d", &dato1 );
scanf( "%d", &dato2 );
printf( "\n" );
```

*Calcular la suma* ⟶

```
suma = dato1 + dato2;
```

*Imprimir el resultado* ⟶

```
printf( "La suma es%10d\n", suma );
```

Uniendo todos los fragmentos finales de código en el orden adecuado, y añadiendo las declaraciones de las variables necesarias, tendremos el programa completo.

```
/** Programa: SumarDosNumeros */
/* Obtener la suma de dos números enteros */

#include <stdio.h>

int main() {
  int dato1, dato2, suma;

  printf( "Dar dos números: " );
  scanf( "%d", &dato1 );
  scanf( "%d", &dato2 );
  printf( "\n" );

  suma = dato1 + dato2;

  printf( "La suma es%10d\n", suma );
}
```

Un ejemplo de la ejecución de este programa sería:

```
Dar dos números: 37 143

La suma es       180
```

### 4.3.2 Ejemplo: Imprimir la silueta de una silla

Aplicaremos la técnica de refinamientos a un programa sencillo en **C±** que imprima de forma esquemática la silueta de una silla usando caracteres normales de escritura, por ejemplo "!" y "=", de la siguiente forma:

```
!
!
======
!    !
!    !
```

La parte ejecutable del programa se planteará inicialmente como una acción única, que trataremos de describir con una frase sencilla; por ejemplo:

*Imprimir la silueta de una silla*

Puesto que no parece fácil programar esta acción con una única sentencia de **C±**, la descompondremos en acciones más simples. De forma intuitiva podemos:

(a) Asociar las acciones componentes a la impresión de diferentes partes de la silla: asiento, patas y respaldo.
(b) Seguir el orden de ejecución impuesto por el hecho de que la impresora ha de ir imprimiendo las líneas de arriba a abajo.

El primer paso de refinamiento será:

*Imprimir la silueta de una silla* ⟶
  *Imprimir la silueta del respaldo*
  *Imprimir la silueta del asiento*
  *Imprimir la silueta de las patas*

Ahora hay que determinar si estas acciones intermedias son ya expresables directamente como sentencias en **C±** o necesitan nuevas descomposiciones. No hay reglas fijas sobre cuándo una acción puede considerarse simple y pasar a escribirla en el lenguaje de programación. En este ejemplo consideraremos simple la escritura de una línea de texto.

De acuerdo con ello podremos considerar como acción simple la impresión del asiento, y refinar:

*Imprimir la silueta del asiento* ⟶

```
printf( "======\n" );
```

La impresión del respaldo se refinaría en la forma:

*Imprimir la silueta del respaldo* ⟶
  *Imprimir parte superior del respaldo*
  *Imprimir parte inferior del respaldo*

y cada una de estas partes conduciría a las mismas sentencias de **C±**:

*Imprimir parte superior o inferior del respaldo* ⟶

```
printf( "!\n" );
```

De forma similar se refinaría la impresión de las patas de la silla. Omitimos este paso para no hacer el ejemplo demasiado prolijo. Finalmente todos estos refinamientos han de ser combinados en un programa único. Reuniendo todas las sentencias en una única secuencia tendremos:

```
printf( "!\n" );
printf( "!\n" );
printf( "======\n" );
printf( "!    !\n" );
printf( "!    !\n" );
```

## 4.4 Aspectos de estilo

Una buena metodología de desarrollo de programas debe atender no sólo a cómo se van refinando las sucesivas acciones, sino a cómo se expresan las acciones finales en el lenguaje de programación. El estilo de redacción del programa en su forma final es algo fundamental para conseguir que sea claro y fácilmente comprensible por parte de quienes hayan de leerlo.

En los apartados que siguen se darán diversas recomendaciones sobre la manera de presentar adecuadamente un programa para facilitar su comprensión. Estas recomendaciones forman parte del *Manual de Estilo* de **C±**.

### 4.4.1 Encolumnado

Un programa aparece como un texto. Dicho texto puede ser visto como un documento técnico, organizado de una manera muy precisa. El estilo de presentación de dicho texto o documento debe destacar claramente su organización en partes.

Un recurso de estilo de presentación es el *encolumnado* o sangrado (*indent*). Ampliando el margen izquierdo para las partes internas del programa se puede conseguir que el texto de un elemento compuesto ocupe una zona aproximadamente rectangular, y que el texto que representa cada uno de sus componentes

ocupe también una zona rectangular dentro de ella. Gráficamente podemos ilustrar esta idea con el esquema de la figura 4.2 en la que se ha marcado el espacio de cada parte como un rectángulo, y se indica con línea doble cuál es el margen izquierdo en cada momento.

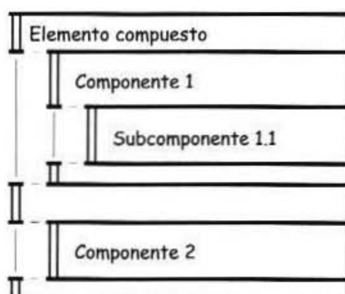



Figura 4.2 Encolumnado de elementos compuestos.

Si aplicamos este recurso de estilo al esquema global de un programa en **C±**, llegaremos a escribir, ya completo, el ejemplo anterior de la siguiente manera:

```
#include <stdio.h>

int main() {
  printf( "!\n" );
  printf( "!\n" );
  printf( "======\n" );
  printf( "!    !\n" );
  printf( "!    !\n" );
}
```

En este caso, la función principal `int main() {...}` es el único elemento compuesto. Los elementos componentes son las sentencias de escritura `printf(...);` que se encolumnan todas ellas dejando cierto margen en blanco a la izquierda. En próximos temas se ilustrará el encolumnado de componentes y subcomponentes cuando se empleen las sentencias estructuradas y los subprogramas.

## 4.4.2 Comentarios. Documentación del refinamiento

Otro recurso utilizable para mejorar la claridad de un programa es el empleo de comentarios. Ya se ha indicado cómo el lenguaje **C±** permite intercalar comentarios en el texto de un programa escribiéndolos entre la pareja de símbolos /* y */.

Aunque el lenguaje permite emplear comentarios con toda libertad, es aconsejable seguir ciertas pautas para facilitar la lectura del programa. Estas pautas corresponden a diferentes clases de comentarios, cada una con un propósito diferente. Entre ellas podemos mencionar:

- Cabeceras de programa
- Cabeceras de sección
- Comentarios-orden
- Comentarios al margen

La *cabecera de programa* tiene como finalidad documentar el programa como un todo. Puede incluir datos de identificación, finalidad, descripción general, etc. Suele presentarse como una "caja" al comienzo del texto del programa, ocupando todo el ancho del listado. A continuación se presenta una posible cabecera para el programa de ejemplo de imprimir la silueta de una silla.

```
/****************************************************************
* Programa: Silla
*
* Autor: Fulano de Tal
*
* Descripción:
*    Este programa imprime de forma esquemática la silueta
*    de una silla usando caracteres normales de la impresora.
****************************************************************/
```

En este ejemplo la "caja" se ha delimitado con asteriscos, y se ha dejado abierta en el lado derecho para facilitar la edición o modificación del texto de la misma.

Las *cabeceras de sección* sirven para documentar partes importantes de un programa relativamente largo. Al igual que la cabecera del programa, se presentan en forma de "caja" al comienzo de la sección correspondientes, ocupando todo el ancho del listado. Un ejemplo trivial sería:

```
/*==========================================
        PARTE EJECUTABLE DEL PROGRAMA
==========================================*/
```

En este ejemplo la "caja" se ha delimitado con signos "=" y se ha dejado abierta a ambos lados.

Los *comentarios-orden* son un elemento metodológico fundamental, y sirven para documentar los refinamientos empleados en el desarrollo del programa. Si analizamos la redacción final del programa de ejemplo de la silla nos encontraremos con que las acciones intermedias que se han ido identificando durante

el proceso de desarrollo por refinamientos sucesivos no aparecen en ninguna parte del texto del programa. Al hacerlo así se ha perdido una información de gran importancia para realizar posteriormente modificaciones en el programa.

Para incluir la información de los pasos de refinamiento en el texto del programa se puede introducir un comentario con la descripción de cada acción intermedia. Estos comentarios tienen, en cierta medida, la categoría de una orden del programa imperativo; en efecto, si el lenguaje de programación lo permitiese, el deseo del programador sería escribir una sentencia que ejecutara dicha acción, pero como no es posible, la acción se descompone en otras más sencillas que dan lugar finalmente a sentencias del programa. El comentario-orden se debe encolumnar tal como se haría con una sentencia del lenguaje que ejecutara la acción deseada. El comentario-orden delimita una acción compuesta, y las acciones componentes se escribirán dejando un mayor margen a la izquierda, tal como se indicó en la sección anterior. En **C±** la acción compuesta puede escribirse entre llaves `{}`. Como ejemplo, se repite aquí la parte ejecutable del programa de imprimir la silueta de la silla, documentando las acciones principales con comentarios-orden y acciones compuestas.

```
/*-- Imprimir el respaldo --*/ {
  printf( "!\n" );
  printf( "!\n" );
}
/*-- Imprimir el asiento --*/ {
  printf( "======\n" );
}
/*-- Imprimir las patas --*/ {
  printf( "!    !\n" );
  printf( "!    !\n" );
}
```

En este ejemplo se han utilizado caracteres fácilmente visibles "`--`" para marcar el principio y el final de los comentarios-orden y distinguirlos de comentarios de otro tipo.

Finalmente mencionaremos la posibilidad de emplear *comentarios al margen*. Estos comentarios sirven para aclarar el significado de ciertas sentencias del programa, que pueden ser difíciles de interpretar al leerlas tal como aparecen escritas en el lenguaje de programación empleado. Una recomendación de estilo es situar estos comentarios hacia la parte derecha del listado, en las mismas líneas que las sentencias que se comentan, y alineados todos a partir de una posición fija, hacia el comienzo del tercio final.

Los comentarios al margen se utilizan muchísimo para explicar el significado de cada variable usada en un programa, poniéndolos en la misma línea en que se declaran. Por ejemplo:

```
float base, altura;  /* dimensiones en cm */
float area;          /* área en cm² */
int volumen;         /* volumen en litros */
int dia;             /* entre 1 y 31 */
```

Como resumen de todas las recomendaciones sobre empleo de comentarios, se muestra a continuación el listado completo del programa de imprimir la silueta de la silla, incluyendo comentarios de todos los tipos indicados. En un programa tan sencillo como éste muchos de los comentarios resultan superfluos y de hecho hacen el programa un tanto engorroso, pero se han incluido para ilustrar la manera de presentar cada clase de comentario.

```
/************************************************************
* Programa: Silla
*
* Descripción:
*    Este programa imprime de forma esquemática la silueta
*    de una silla usando caracteres normales de la impresora.
************************************************************/

/*============================================
         DIRECTIVA DE COMPILACIÓN
============================================*/
#include <stdio.h>

/*============================================
         PARTE EJECUTABLE DEL PROGRAMA
============================================*/
int main() {
  /*-- Imprimir el respaldo --*/ {
    printf( "!\n" );
    printf( "!\n" );
  }
  /*-- Imprimir el asiento --*/ {
    printf( "=====\n" );
  }
  /*-- Imprimir las patas --*/ {
    printf( "!   !\n" );
    printf( "!   !\n" );
  }
}
```

### 4.4.3 Elección de nombres

Otro aspecto de estilo, fundamental para la claridad de un programa, es una *elección* correcta de los nombres o identificadores utilizados para designar sus diferentes elementos. Los nombres que tenga que inventar el programador deben ser elegidos con un criterio nemotécnico, de manera que recuerden fácilmente el significado de los elementos nombrados.

Para evidenciar la importancia de usar nombres adecuados, compararemos dos redacciones de un mismo programa. La primera, con nombres carentes de significado sería:

```
#include <stdio.h>

int main() {
  int x, y, z;

  scanf( "%d", &x );
  scanf( "%d", &y );
  z = x * y;
  printf( "Area = %10d" ,z );
}
```

La segunda redacción del programa, con nombres significativos, podría ser:

```
#include <stdio.h>

int main() {
  int base, altura, area;

  scanf( "%d", &base );
  scanf( "%d", &altura );
  area = base * altura;
  printf( "Area = %10d",area );
}
```

Comparando ambas redacciones es muy fácil darse cuenta de la ventaja de la segunda, cuyo significado puede adivinarse fácilmente incluso con la total ausencia de comentarios explicativos en el programa.

Para que los nombres o identificadores resulten significativos hay que procurar que tengan la categoría gramatical adecuada al elemento nombrado. En concreto:

- Los valores (constantes, variables, etc.) deben ser designados mediante sustantivos.
- Las acciones (procedimientos, etc.) deben ser designadas con verbos.

- Los tipos deben ser designados mediante nombres genéricos.

En programas largos con muchos identificadores resulta a veces difícil inventar nombres significativos que distingan fácilmente entre valores y tipos. Un recurso de estilo puede ser construir los identificadores de tipo usando sistemáticamente el prefijo `Tipo` o similar en todos ellos; por ejemplo `TipoLongitud` o bien `T_longitud`.

▮ NOTA: En los lenguajes C y C++ es también habitual usar sufijos en lugar de prefijos para distinguir diferentes categorías de identificadores. En particular el sufijo `_t` para los nombres de tipos, incluso en algún tipo predefinido como `wchar_t` en C++.

Al usar verbos para nombrar acciones conviene emplearlos de manera uniforme utilizando siempre el mismo tiempo gramatical. En inglés suelen aparecer en imperativo. En español resulta quizá más natural emplear sistemáticamente el infinitivo. De hecho en inglés el imperativo y el infinitivo coinciden en muchos casos. Ejemplos:

| Término inglés | Imperativo | Infinitivo |
|---|---|---|
| writeNumber | escribeNumero | escribirNumero |
| read_date | lee_fecha | leer_fecha |
| go_to_origin | ve_al_origen | ir_al_origen |

A continuación se presenta un ejemplo de programa completo, incluyendo identificadores de diferentes clases elegidos según las recomendaciones anteriores. En este ejemplo se ha supuesto que existe un módulo llamado `Fechas` que define procedimientos para manipular datos de tipo fecha. El estilo seguido para elegir los nombres inventados por el programador ha sido:

- Los nombres de operaciones (procedimientos y programa principal) son verbos en infinitivo:

```
CalcularDias  LeerFecha  EscribirFecha
```

- Los nombres de valores ( funciones y variables ) son sustantivos:

```
DiasEntre  Hoy   fechaCumple
fechaHoy   dias
```

- Los nombres de tipo empiezan por el prefijo `T_`:

```
T_fecha
```

```
/************************************************************
* Programa: CalcularDias
*
* Descripción:
*   Este programa calcula los días que faltan para el
*   cumpleaños de una persona.
************************************************************/
#include <stdio.h>
#include "fechas.h"

/*=====================================================
 En el módulo fechas.h están definidos:

  T_fecha = Tipo de valor FECHA
  LeerFecha = Procedimiento para leer una fecha
  EscribirFecha = Procedimiento para escribir una fecha
  DiasEntre = Función para calcular los dias que hay
              entre dos fechas
  Hoy = Variable en la que se mantiene actualizada la
        fecha de hoy
=====================================================*/

int main() {

  T_fecha fechaCumple;  /* cumpleaños */
  T_fecha fechaHoy;     /* fecha de hoy */
  int dias;             /* días que faltan */

  /*-- Obtener la fecha del cumpleaños --*/ {
    printf( "¿Cuál es tu próximo cumpleaños? " );
    LeerFecha( fechaCumple );
  }
  /*-- Obtener la fecha de hoy --*/ {
    fechaHoy = Hoy;
  }
  /*-- Calcular los días que faltan para el cumpleaños --*/ {
    dias = DiasEntre( fechaHoy, fechaCumple );
  }
  /*-- Imprimir el resultado --*/ {
    printf( "\nFaltan%4d", dias);
    printf( " días para tu cumpleaños" );
  }
}
```

### 4.4.4 Uso de letras mayúsculas y minúsculas

Los lenguajes de programación que permiten distinguir entre letras *mayúsculas* y *minúsculas* facilitan la construcción de nombres en programas largos, en que es preciso inventar un gran número de ellos.

Por ejemplo, para marcar claramente a qué clase de elemento del programa se está refiriendo un nombre, se puede adoptar el criterio de que los identificadores de ciertas clases de elementos empiecen siempre con mayúscula, y otros siempre con minúscula, o que algunos muy especiales se escriban todo en mayúsculas. En el ejemplo del apartado anterior se han seguido las siguientes reglas de estilo para escribir los identificadores inventados por el programador:

- Los nombres de tipos, procedimientos y funciones empiezan por mayúscula:

  ```
  CalcularDias  LeerFecha  EscribirFecha
  DiasEntre     Hoy
  ```

- Los nombres de variables y constantes empiezan por minúscula:

  ```
  fechaCumple  fechaHoy  dias
  ```

- Los nombres que son palabras compuestas usan mayúsculas intercaladas al comienzo de cada siguiente palabra componente:

  ```
  TipoLongitud  TipoFecha    fechaHoy
  CalcularDias  LeerFecha    EscribirFecha
  DiasEntre     fechaCumple
  ```

En cualquier caso hay que limitar mucho el empleo de nombres escritos totalmente en mayúsculas, ya que en general hacen más pesada la lectura del texto del programa. Sólo deberían escribirse así elementos que han de destacar entre los demás.

Compárese el siguiente programa, escrito todo en mayúsculas, con el ejemplo correspondiente del apartado 4.4.3. La lectura resulta ahora mucho más difícil.

```
#INCLUDE <STDIO.H>

INT MAIN() {
  INT BASE, ALTURA, AREA;

  SCANF("%D",&BASE);
  SCANF("%D",&ALTURA);
  AREA = BASE * ALTURA;
  PRINTF("AREA = %10D",AREA);
}
```

▌NOTA: El programa anterior no es válido en **C±** ni en C o C++, que requieren escribir ciertas palabras en minúsculas. Se ha presentado sólo como ejemplo de la dificultad de lectura que conlleva el uso excesivo o exclusivo de mayúsculas.

### 4.4.5 Constantes con nombre

La posibilidad de declarar *constantes con nombres* simbólicos puede aprovecharse para mejorar la claridad del programa. En lugar de usar directamente valores numéricos en las expresiones de algunos cálculos, puede resultar ventajoso definir determinados coeficientes o factores de conversión con un nombre simbólico que tenga un buen significado nemotécnico, y usar la constante con ese nombre en los cálculos. Por ejemplo, para transformar una longitud de pulgadas a centímetros, en lugar de escribir:

```
longitudCm = longitudPul * 2.54
```

se podría poner

```
const float cmPorPulgada = 2.54;
. . . .
longitudCm = longitudPul * cmPorPulgada
```

Esta segunda forma resultará más significativa para los que no recuerden de memoria ese factor de conversión. Además se tiene una ventaja adicional en el caso de que un mismo valor constante se use en varios puntos del programa; al definirlo como constante con nombre el valor numérico particular se escribe sólo una vez, en la definición, en lugar de hacerlo tantas veces como se use, y así se reducen las posibilidades de cometer errores de escritura.

Otra forma ventajosa de usar el mecanismo de definición de constantes con nombre se da en el caso de que el comportamiento de un programa venga dado en función de ciertos valores generales, fijos, pero que quizá fuera interesante cambiarlos en el futuro. Este tipo de valores se denominan a veces *parámetros del programa*, y es conveniente que su valor aparezca escrito sólo una vez en un lugar destacado del programa. Por ejemplo, un programa para reformar un texto antes de enviarlo a la impresora puede tener como parámetros generales el ancho de la línea de escritura, o el tamaño de la página. Podríamos escribir, al comienzo del programa:

```
/*================================================
              PARAMETROS GENERALES
================================================*/
  const int ANCHO_LINEA = 72;
  const int LINEAS_PAGINA = 60;
```

De esta manera queda perfectamente destacada la parte del programa que hay que modificar si se quieren cambiar las dimensiones útiles de la hoja impresa.

## 4.5 Ejemplos de programas

A continuación se desarrollan algunos programas sencillos mediante refinamientos sucesivos.

### 4.5.1 Ejemplo: Imprimir la figura de un árbol de navidad

Se trata de escribir un programa que imprima la silueta de un árbol de navidad convencional, según aparece en el siguiente listado resultado de la impresión:

```
    *
   ***
  *****
   ***
  *****
 *******
  *****
 *******
*********
    *
    *
    *
  *****
```

Para estructurar el programa trataremos de identificar las diferentes partes de la figura del árbol. Podemos reconocer la copa (formada por tres pisos de ramas), el tronco y la base. Teniendo en cuenta que las distintas partes han de imprimirse de arriba a abajo, organizaremos los primeros pasos de refinamiento:

*Imprimir arbol* ⟶
  *Imprimir copa*
  *Imprimir tronco*
  *Imprimir base*

*Imprimir copa* ⟶
  *Imprimir primeras ramas*
  *Imprimir segundas ramas*
  *Imprimir terceras ramas*

La impresión de cada parte se consigue directamente con sentencias de escritura sencillas. Podemos pasar ya a escribir el programa en **C±**, documentándolo de acuerdo con las reglas de estilo propuestas. El programa completo aparece a continuación.

```
/*****************************************************
* Programa: Arbol
*
* Descripción:
*   Este programa imprime la silueta de un árbol
*   de navidad, hecha con asteriscos
*****************************************************/
#include <stdio.h>

int main() {
  /*-- Imprimir copa --*/ {
    /*-- Imprimir primeras ramas --*/ {
      printf( "    *\n" );
      printf( "   ***\n" );
      printf( "  *****\n" );
    }
    /*-- Imprimir segundas ramas --*/ {
      printf( "   ***\n" );
      printf( "  *****\n" );
      printf( " *******\n" );
    }
    /*-- Imprimir terceras ramas --*/ {
      printf( "  *****\n" );
      printf( " *******\n" );
      printf( "*********\n" );
    }
  }
  /*-- Imprimir tronco--*/ {
    printf( "    *\n" );
    printf( "    *\n" );
    printf( "    *\n" );
  }
  /*-- Imprimir base -- */ {
    printf( "  *****\n" );
  }
}
```

### 4.5.2 Ejemplo: Calcular el costo de las baldosas

Se trata de calcular el costo total de las baldosas necesarias para cubrir el suelo de una habitación rectangular. Se supone que las baldosas son cuadradas. El programa lee como dato el lado de las baldosas, en centímetros, y las dimensiones de la habitación rectangular en metros. También se suministra como dato el precio unitario de cada baldosa.

El programa calcula cuántas baldosas hay que colocar a lo largo de cada dimensión de la habitación, incluyendo contar una baldosa más si no es un número entero y hay que romper algunas baldosas para cubrir exactamente hasta el borde. A continuación se calcula el número total de baldosas y se multiplica por el precio de cada una.

De forma abreviada, los primeros pasos de descomposición conducen a:

*Calcular el costo de baldosas* ⟶
 *Leer los datos*
 *Calcular el número de baldosas*
 *Calcular el coste total*
 *Imprimir el resultado*

*Calcular el número de baldosas* ⟶
 *Calcular las baldosas a lo largo*
 *Calcular las baldosas a lo ancho*
 *Calcular el número total de baldosas*

El programa completo, debidamente documentado, aparece en el siguiente listado:

```
/**********************************************************
* Programa: Baldosas
*
* Descripción:
*   Este programa calcula el costo de las baldosas
*   necesarias para cubrir una habitación rectangular
**********************************************************/
#include <stdio.h>

int main() {
  int largo, ancho; /* Dimensiones de la habitación en m */
  int lado;         /* Lado de la baldosa en cm */
  int nLargo;       /* Número de baldosas a lo largo */
  int nAncho;       /* Número de baldosas a lo ancho */
  int baldosas;     /* Número total de baldosas */
  float precio;     /* Precio de cada baldosa */
```

```
  float coste;        /* Coste total */

  /*-- Leer los datos --*/ {
    printf( "Dar el tamaño de la habitación, en m\n" );
    printf( "¿Largo, ancho? " );
    scanf( "%d,%d",&largo,&ancho);
    printf( "¿Lado de la baldosa, en cm? " );
    scanf( "%d", &lado );
    printf( "¿Precio de cada baldosa, en euros? ");
    scanf( "%f", &precio );
    /*--Calcular el número de baldosas--*/ {
      /*--Calcular las baldosas a lo largo, por exceso--*/ {
        nLargo = (largo*100 + lado - 1) / lado;
      }
      /*--Calcular las baldosas a lo ancho, por exceso--*/ {
        nAncho = (ancho*100 + lado - 1) / lado;
      }
      /*--Calcular el número total de baldosas--*/ {
        baldosas = nLargo * nAncho;
      }
    }
    /*--Calcular el coste total--*/ {
      coste = baldosas * precio;
    }
    /*--Imprimir el resultado--*/ {
      printf( "Total %5d baldosas\n", baldosas );
      printf( "Coste %8.2f euros\n", coste );
    }
  }
}
```

El resultado de una posible ejecución es el siguiente:

```
Dar el tamaño de la habitación, en m
¿Largo, ancho? 4, 6
¿Lado de la baldosa, en cm? 30
¿Precio de cada baldosa, en euros? 0.56
Total   280 baldosas
Coste   156.80 euros
```

### 4.5.3 Ejemplo: Calcular los días entre dos fechas

Este ejemplo consiste en calcular la diferencia en días entre dos fechas. Para simplificar, el cálculo se hace de forma aproximada, contando todos los meses a razón de 30 días cada uno, y los años completos siempre con 365 días.

El cálculo se hace en dos partes. Primero se calculan para cada fecha los días transcurridos desde el comienzo de su año. Luego se calcula la diferencia entre fechas, pasando la diferencia en años a días y acumulando la diferencia en días dentro del año. Los pasos de descomposición son relativamente sencillos:

*Calcular la diferencia de fechas* ⟶
*Leer las fechas*
*Calcular la diferencia*
*Imprimir el resultado*

*Calcular la diferencia* ⟶
*Calcular los días desde principio de año*
*Calcular la diferencia total de días*

El programa completo aparece en el siguiente listado:

```
/********************************************************
* Programa: DiferenciaEnDias
*
* Descripción:
*   Este programa calcula los días entre dos
*   fechas, de forma aproximada, contando todos
*   meses de 30 días, y los años de 365
********************************************************/
#include <stdio.h>

int main() {
  int dia1, mes1, anno1;      /* primera fecha */
  int dia2, mes2, anno2;      /* segunda fecha */
  int diasFecha1, diasFecha2; /* días desde inicio del año */
  int diferencia;             /* diferencia en días */

  /*-- Leer las fechas --*/ {
    printf( "¿Primera fecha (dd,mm,aaaa)? " );
    scanf( "%d,%d,%d", &dia1, &mes1, &anno1 );
    printf( "¿Segunda fecha (dd,mm,aaaa)? " );
    scanf( "%d,%d,%d", &dia2, &mes2, &anno2 );
  }
  /*-- Calcular la diferencia --*/ {
    /*--Calcular los días desde principio del año--*/ {
      diasFecha1 = (mes1 - 1)*30 + dia1;
      diasFecha2 = (mes2 - 1)*30 + dia2;
    }
    /*--Calcular la diferencia total en días--*/ {
      diferencia = (anno2 - anno1)*365
                   + diasFecha2 - diasFecha1;
    }
```

```
  }
  /*--Imprimir el resultado--*/ {
    printf( "Desde %2d/%2d/%4d\n", dia1, mes1, anno1 );
    printf( "hasta %2d/%2d/%4d\n", dia2, mes2, anno2 );
    printf( "hay %5d días\n", diferencia);
  }
}
```

El resultado de una posible ejecución del programa es el siguiente:

```
¿Primera fecha (dd,mm,aaaa)? 20,3,2009
¿Segunda fecha (dd,mm,aaaa)? 14,1,2011
Desde 20/ 3/2009
hasta 14/ 1/2011
hay   664 días
```

# Tema 5

# Estructuras Básicas de la Programación Imperativa

Este tema se dedica a introducir las estructuras básicas de la programación imperativa: Secuencia, Selección e Iteración, indicando la manera de realizar dichas estructuras en **C±**. Aunque existen otras estructuras, que se introducen en temas posteriores, en este tema se ha preferido insistir fundamentalmente en los conceptos subyacentes en estas estructuras básicas, prescindiendo de los aspectos específicos de un lenguaje particular.

Inicialmente se presentan sólo las estructuras básicas IF-THEN-ELSE y WHILE, y su codificación en **C±**. Posteriormente se mencionan las estructuras particulares IF-ELSIF-ELSE y FOR derivadas de las anteriores, justificándolas por la comodidad de su uso.

## 5.1 Programación estructurada

La *programación estructurada* es una metodología de programación que fundamentalmente trata de construir programas que sean fácilmente comprensibles. Un programa no solamente debe funcionar correctamente, sino que además debe estar escrito de manera que se facilite su comprensión posterior.

Normalmente en todos los programas se tienen que realizar algunas correcciones o modificaciones después de trascurrido un cierto tiempo. En ese momento se trata de evitar lo que a veces llega a suceder, desgraciadamente: si el programa no está claramente escrito, ni el mismo programador que construyó el programa es capaz de entender cómo funciona.

Esta metodología está basada en la técnica de desarrollo de programas por *refinamientos sucesivos*, tal como se ha expuesto en el tema anterior. Inicialmente, se plantea la operación global a realizar por el programa, y se descompone en otras más sencillas. A su vez, estas últimas vuelven a ser descompuestas nuevamente en otras todavía más elementales. Este proceso de descomposición continúa hasta que todo se puede escribir utilizando las estructuras básicas disponibles en el lenguaje de programación que se está empleando.

### 5.1.1 Representación de la estructura de un programa

La estructura de los programas imperativos se representa tradicionalmente mediante *diagramas de flujo*, llamados en inglés "*flow-chart*". Estos diagramas contienen dos elementos básicos, correspondientes a *acciones* y *condiciones* (figura 5.1). Las acciones se representan mediante rectángulos, y las condiciones mediante rombos. Las condiciones equivalen a preguntas a las que se puede responder "Sí" o "No".

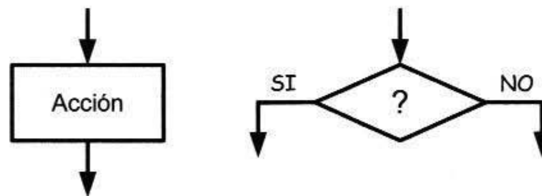



Figura 5.1 Símbolos de acción y condición.

El *flujo de control* durante la ejecución del programa se refleja mediante líneas o vías que van de un elemento a otro. Las acciones tienen una sola vía de entrada o comienzo y una de terminación o salida. Las condiciones tienen una vía de entrada, y dos vías de salida marcadas con "Sí" y "No". Durante la ejecución, cuando el flujo llega a la entrada de una acción, la acción se realiza y el flujo se dirige a su salida. Cuando se llega a la entrada de una condición, la condición se evalúa, y si resulta ser cierta se continúa por la salida "Sí", mientras que si es falsa se continúa por la salida "No". La figura 5.2 contiene un ejemplo sencillo de diagrama de flujo. En esta figura se indica también cómo un fragmento del diagrama, que tenga un solo punto de entrada y uno de salida, puede ser visto globalmente como una acción única, pero compuesta.

La parte de diagrama de flujo en el interior de una acción compuesta constituye la estructura o esquema de dicha acción. La programación estructurada recomienda descomponer las acciones usando las estructuras más sencillas posibles. Entre ellas se reconocen tres estructuras básicas, que son: *Secuencia*,

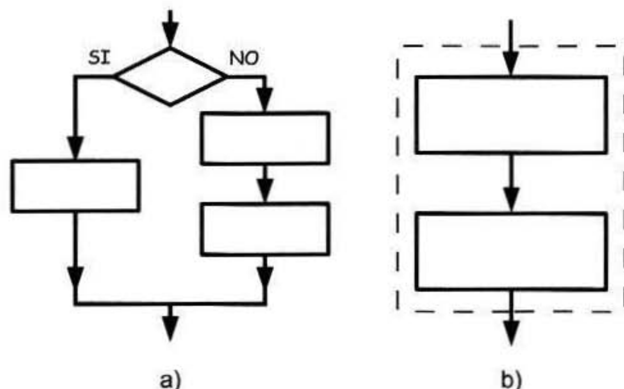



Figura 5.2 Ejemplo de diagrama de flujo (a) y acción compuesta (b).

*Selección* e *Iteración*. Estas tres estructuras están disponibles en todos los lenguajes modernos de programación imperativa en forma de sentencias del lenguaje. Combinando unos esquemas con otros se pueden llegar a construir programas con una estructura tan complicada como sea necesario.

### 5.1.2 Secuencia

La estructura más sencilla para emplear en la descomposición es utilizar una *secuencia* de acciones o partes que se ejecutan de forma sucesiva. En la figura 5.3 se muestra una secuencia de acciones.

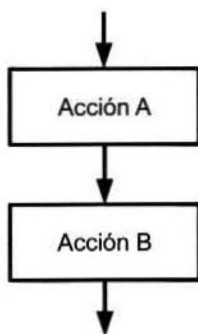



Figura 5.3. Secuencia.

La estructura secuencial ya ha sido utilizada en los ejemplos realizados en los temas anteriores. Todos ellos han sido resueltos como una secuencia de sentencias elementales del lenguaje.

### 5.1.3 Selección

La estructura de *selección* consiste en ejecutar una acción u otra, dependiendo de una determinada condición que se analiza a la entrada de la estructura. En la figura 5.4 se puede ver la estructura de selección. Si la condición analizada <?> da como resultado "Sí" se realiza la acción A y si el resultado es "No" se realiza la acción B. Como se puede observar sólo tiene una única entrada y una única salida.

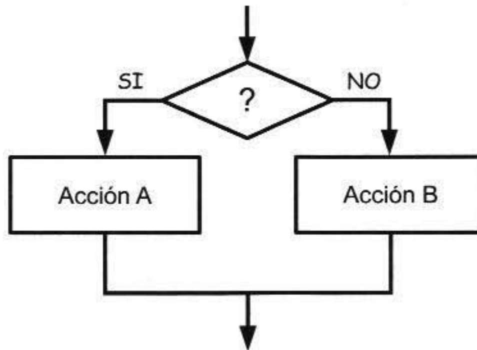



Figura 5.4 Selección.

### 5.1.4 Iteración

La *iteración* es la repetición de una acción mientras que se cumpla una determinada condición. La estructura de iteración más general es aquella en que la condición se analiza a la entrada de la estructura y antes de iniciar cada nueva repetición. En la figura 5.5 se muestra esta estructura de iteración. Cada vez que se analiza la condición <?> se pueden dar dos resultados. Si el resultado es "Sí" se ejecuta nuevamente la acción. Una vez ejecutada la acción se vuelve a analizar la condición <?>. En el momento que el resultado es "No" se alcanza el punto final de la estructura. También en este caso sólo existe un punto de entrada y un punto de salida.

Puesto que el flujo de ejecución vuelve hacia atrás siguiendo un camino cerrado, la estructura de iteración se denomina también *bucle*.

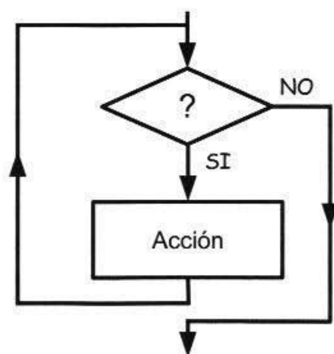



Figura 5.5 Iteración.

### 5.1.5 Estructuras anidadas

Cualquier parte o acción del programa puede a su vez estar constituida por cualquiera de las estructuras descritas. Por tanto, el anidamiento entre ellas puede ser tan complejo como sea necesario.

Mediante la técnica de *refinamientos sucesivos* se definen inicialmente las estructuras más externas del programa y en los pasos sucesivos se va detallando la estructura de cada acción compuesta. Este proceso finalmente da lugar a que todo el programa quede escrito utilizando las estructuras básicas descritas en este apartado, anidadas unas dentro de otras.

## 5.2 Expresiones condicionales

Para poder utilizar las estructuras de selección e iteración es necesario expresar las condiciones <?> que controlan ambas estructuras. Esto se realiza mediante la construcción de expresiones condicionales. Estas expresiones sólo pueden dar como resultado dos valores: "Sí" (cierto), cuando se cumple la condición de la expresión, y "No" (falso), en caso de que no se cumpla.

Una primera forma de construir *expresiones condicionales* es mediante el empleo de operadores de comparación en expresiones aritméticas. Estos operadores permiten realizar comparaciones entre dos valores del mismo tipo. Es muy importante resaltar que en el *Manual de Estilo* de esta asignatura no se permite la comparación entre elementos de distinto tipo (por ejemplo: enteros con caracteres, reales con enteros, fechas con colores, etc.). Las operaciones de comparación disponibles y sus operadores en **C±** son las siguientes:

| Comparación | Símbolo matemático | Operador **C±** |
|---|---|---|
| Mayor que | $>$ | `>` |
| Mayor o igual que | $\geq$ | `>=` |
| Menor que | $<$ | `<` |
| Menor o igual que | $\leq$ | `<=` |
| Igual a | $=$ | `==` |
| Diferente a | $\neq$ | `!=` |

Los símbolos de operadores **C±** con dos caracteres deben escribirse precisamente en ese orden, y sin espacio en blanco entre ellos.

Veamos un ejemplo. Sean las variables declaradas siguientes:

```
int largo, ancho;
float presion, temperatura;
char letra, modelo;
```

Con los operadores de comparación se pueden formar expresiones condicionales tales como las siguientes:

```
largo > 5
ancho == largo
presion <= 23.5
modelo = 'Z'
letra != modelo
presion != temperatura
```

Con los operadores de comparación sólo es posible realizar una única comparación entre dos valores. Sin embargo, es bastante normal que las condiciones sean más complejas. Pueden construirse condiciones que impliquen a más de dos valores como condiciones compuestas de varias condiciones simples.

Las condiciones compuestas se construyen como *expresiones lógicas*. Cada término de una expresión lógica podrá ser una expresión condicional simple. Las operaciones lógicas entre dos expresiones simples $E_1$, $E_2$ y los correspondientes operadores disponibles en **C±** son los siguientes:

| Operación lógica | Símbolo matemático | Operador **C±** |
|---|---|---|
| Conjunción ($E_1$ y $E_2$) | $\wedge$ | `&&` |
| Disyunción ($E_1$ o $E_2$) | $\vee$ | `||` |
| Negación (no $E_1$) | $\neg$ | `!` |

La operación de conjunción `E1 && E2` da resultado cierto si tanto `E1` como `E2` son ciertos. En el lenguaje **C±** para evaluar la operación de conjunción `&&` siempre se empieza por evaluar la expresión simple `E1` del primer operando, y si su resultado es falso ya no se evalúa la expresión `E2` del segundo operando.

Está claro que el resultado de la conjunción ya no depende del valor de `E2` y será siempre falso. Por este motivo se dice que el operador `&&` se evalúa *en cortocircuito*. Esta misma regla se aplica en el caso de realizar la conjunción de $n$ expresiones `E1 && E2 && E3 && ... && E`$n$ y sólo se continuará evaluando una nueva expresión `E`$i$ cuando todas las anteriores hayan sido ciertas.

La operación de disyunción `E1 || E2` da resultado cierto si una de las dos, `E1` o `E2` , o ambas, son ciertas. También el operador `||` se evalúa *en cortocircuito* y en la disyunción de $n$ expresiones `E1 || E2 || E3 || ... || E`$n$ sólo se continua evaluando una nueva expresión `E`$i$ cuando todas las anteriores hayan sido falsas.

El operador `!` se aplica a un solo término y niega el resultado de dicho término. Este *operador unario* se utiliza cuando una condición queda expresada de manera más sencilla como complemento de otra. Por supuesto, el operador `!` sólo se puede utilizar para negar elementos de tipo Sí(cierto)/No(falso).

Con estos nuevos operadores es posible construir condiciones complejas tales como las siguientes:

```
(largo > 5) && (ancho < 7)
(modelo == 'A') || (modelo == 'Z')
! (letra == 'Q')
(temperatura <= 123.7) && (presion < 12.3)
```

Su significado se puede deducir fácilmente. La razón del empleo de paréntesis es indicar el orden preciso de ejecución de las operaciones: primero las comparaciones y posteriormente las operaciones lógicas. El empleo de paréntesis en cualquier lenguaje de programación evita la ambigüedad en las expresiones. Por el contrario, un uso excesivo de paréntesis para detallar el orden estricto de ejecución puede llegar a resultar algo farragoso y disminuir la claridad del programa.

Todos los lenguajes tienen definido un orden por defecto para la evaluación de los operadores en las expresiones complejas. Así, como se verá a continuación, en el lenguaje **C±** se realizan primero las operaciones de comparación y posteriormente las operaciones lógicas. Por tanto, en las expresiones anteriores se podría prescindir de todos los paréntesis salvo para la negación de la tercera.

La complejidad de las expresiones puede ser tan grande como sea necesario; el número de términos lógicos que pueden combinarse es ilimitado. Además, cada valor numérico se puede obtener mediante una *expresión aritmética*. Por ejemplo, son expresiones condicionales válidas las siguientes:

```
(largo < 3) && (ancho < 9) && (largo*ancho < 25)
!((letra == 'Q' || letra == 'Z'))
(3.5*temperatura - presion/5.6) < 54.6
```

En la evaluación de estas expresiones complejas el orden por defecto que se sigue viene fijado por el nivel de prioridad que tienen asignadas las distintas operaciones. Es decir, cada operador tiene una prioridad determinada. Si no se utilizan paréntesis, el *orden de evaluación* en el lenguaje **C±** es el siguiente:

1. Operadores Unarios: `! + -`
2. Operadores Multiplicativos: `* / %`
3. Operadores Aditivos: `+ -`
4. Operadores de Comparación: `> >= < <=`
5. Operadores de Igualdad: `== !=`
6. Operador de Conjunción: `&&`
7. Operador de Disyunción: `||`

Los operadores unarios `+` y `-` permiten indicar el signo del único operando al que preceden y no se deben confundir con los operadores aditivos `+` y `-` que necesitan dos operandos para calcular su suma o su resta respectivamente.

Dentro del mismo nivel de prioridad las operaciones se evalúan en el orden en que están escritas en cada expresión concreta, de izquierda a derecha. Como ejemplo escribiremos ahora sin paréntesis las anteriores expresiones:

```
largo < 3 && ancho < 9 && largo*ancho < 25
!letra == 'Q' || letra == 'Z'
3.5*temperatura - presion/5.6 < 54.6
```

Siguiendo los criterios de evaluación indicados, estas expresiones serían equivalentes a las que se indican a continuación, en las que se ha marcado con paréntesis el orden de la evaluación por defecto:

```
(largo < 3) && (ancho < 9) && ((largo*ancho) < 25)
((!letra) == 'Q') || (letra == 'Z')
((3.5*temperatura) - (presion/5.6)) < 54.6
```

Con estas expresiones sólo se trata de mostrar las reglas de evaluación por defecto, dado que como se puede observar, la segunda expresión está mal construida, ya que no tiene ningún sentido hacer la negación de un carácter `!letra` y después comparar el resultado con el carácter `'Q'`. Por otro lado, es interesante resaltar que las expresiones primera y tercera son completamente correctas y equivalentes a las expresiones sin ningún paréntesis o bien a las expresiones originales con paréntesis para explicitar el orden de ejecución: primero comparar y luego realizar las operaciones lógicas.

Para formalizar estos conceptos, las reglas BNF que definen cómo se pueden escribir expresiones aritméticas, condicionales y lógicas en el lenguaje **C±** son las siguientes:

*Expresión* **::=** *Expresión_OR* **{** *Operador_OR Expresión_OR* **}**

*Expresión_OR* **::=** *Expresión_AND*
  **{** *Operador_AND Expresión_AND* **}**

*Expresión_AND* **::=** *Expresión_igualdad*
  **[** *Operador_igualdad Expresión_igualdad* **]**

*Expresión_igualdad* **::=** *Expresión_numérica*
  **[** *Operador_comparación Expresión_numérica* **]**

*Expresión_numérica* **::=** *Término* **{** *Operador_sumador Término* **}**

*Término* **::=** *Factor* **{** *Operador_multiplicador Factor* **}**

*Factor* **::=** `+` *Factor* **|** `-` *Factor* **|** `!` *Factor* **|** *Identificador_de_variable* **|**
  *Identificador_de_constante* **|** *Valor_constante* **|** `(`*Expresión* `)`

*Operador_OR* **::=** `||`

*Operador_AND* **::=** `&&`

*Operador_igualdad* **::=** `==` **|** `!=`

*Operador_comparación* **::=** `>` **|** `>=` **|** `<` **|** `<=`

*Operador_sumador* **::=** `+` **|** `-`

*Operador_multiplicador* **::=** `*` **|** `/` **|** `%`

Como resumen de este apartado, dentro del *Manual de Estilo* que utilizaremos en esta asignatura se establecen las siguientes normas:

1. Es aconsejable utilizar paréntesis adicionales para evitar cualquier ambigüedad o dificultad de interpretación de la expresión.
2. Es aconsejable utilizar paréntesis adicionales siempre que se mejore la claridad de la expresión.
3. No es aconsejable utilizar paréntesis adicionales en aquellas expresiones que, aprovechando los niveles de prioridad por defecto del lenguaje, estén ampliamente consensuadas y no planteen ninguna duda en su interpretación.
4. No están permitidas las comparaciones entre elementos de distintos tipos.
5. Los operadores lógicos ( `&&` y `||` )sólo se pueden utilizar con elementos de tipo Sí(cierto)/No(falso)

## 5.3 Estructuras básicas en C±

En este apartado se describen las sentencias de **C±** que corresponden a las estructuras básicas introducidas anteriormente. Además se mencionan algunas posibilidades adicionales de estas sentencias, que permiten algunas variantes de las estructuras básicas, y que simplifican la escritura de ciertos programas.

### 5.3.1 Secuencia

Esta estructura ya ha sido introducida en el tema 2 y utilizada en todos los programas y ejemplos realizados hasta ahora. Para programar una *secuencia* de acciones en **C±** se escriben las sentencias que forman la secuencia de acciones una tras otra. Así, la secuencia de la figura 5.3 se escribe simplemente de la siguiente manera:

```
Acción A
Acción B
```

Formalmente, según se vio en el tema 2, la sintaxis de la estructura secuencia es:

*Secuencia_de_sentencias* **::=** **{** *Sentencia* **}**

### 5.3.2 Sentencia IF

En **C±** la estructura de *selección* de la figura 5.4 se programa como una sentencia IF que tiene el siguiente formato:

```
if ( Condición ) {
   Acción A
} else {
   Acción B
}
```

La ejecución de la sentencia `if` consiste en evaluar la expresión de *Condición*, y a continuación ejecutar o bien la *Acción A* (si se cumple la condición), o bien la *Acción B* (si la condición no se cumple). Las palabras clave `if` y `else` separan las distintas partes de la sentencia. Por ejemplo:

```
if (largo > ancho) {
   ladoMayor = largo;
} else {
   ladoMayor = ancho;
}
```

En ocasiones no es necesario ejecutar nada cuando la *Condición* no se cumple. La estructura en estos casos queda reducida a la que se muestra en la figura 5.6.

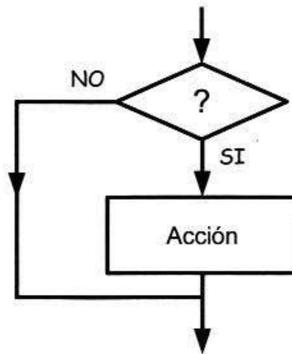



Figura 5.6 Selección simple.

El formato de la sentencia en **C±** es ahora el siguiente:

```
if ( Condición ) {
  Acción
}
```

En este caso se ejecuta la *Acción* cuando la expresión *Condición* se cumple y en caso contrario no se ejecuta nada. Así, el ejemplo anterior se puede programar también como:

```
ladoMayor = ancho;
if (largo > ancho) {
  ladoMayor = largo;
}
```

Es bastante frecuente realizar selecciones que dan lugar a más de dos posibilidades. Por ejemplo, ciertas tarifas pueden ser diferentes según la edad:

| | |
|---|---|
| Niños de 0 a 6 años | Gratis |
| Jóvenes de 6 hasta 18 años | 50 % |
| Adultos de 18 hasta 65 años | 100 % |
| Jubilados de 65 años en adelante | 25 % |

Tal como se indicó anteriormente, es posible anidar varias estructuras de selección unas dentro de otras. Así, se podría realizar esta selección de la siguiente forma:

```
if (edad < 6) {
   tarifa = 0.0;
} else {
   if (edad < 18) {
      tarifa = 0.5;
   } else {
      if (edad < 65) {
         tarifa = 1.0;
      } else {
         tarifa = 0.25;
      }
   }
}
```

Si, como en este caso, la evaluación de las condiciones se hace *en cascada*, atendiendo a una de ellas sólo si todas las anteriores han sido falsas, se puede simplificar la escritura en **C±** de la sentencia IF eliminando las llaves `{...}` de las ramas `else` para expresar directamente una cadena de selecciones. El formato general se representa gráficamente en la figura 5.7.

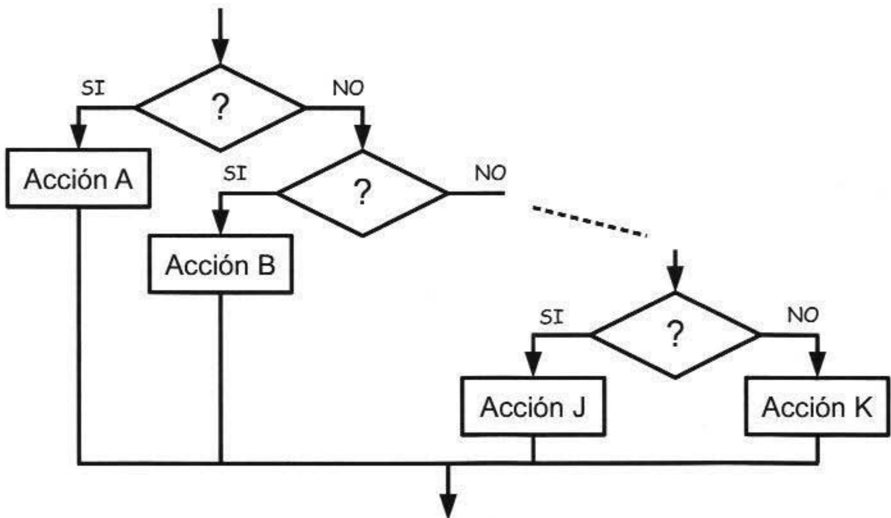



Figura 5.7 Selección en cascada.

El formato de la sentencia `if` para la *selección en cascada* es el siguiente:

```
if ( Condición 1 ) {
   Acción A
} else if ( Condición 2 ) {
   Acción B
   ....
```

```
} else if ( Condición N ) {
  Acción J
} else {
  Acción K
}
```

Con esta estructura la selección anterior se escribe en **C±** de la siguiente manera:

```
if (edad < 6) {
  tarifa = 0.0;
} else if (edad < 18) {
  tarifa = 0.5;
} else if (edad < 65) {
  tarifa = 1.0;
} else {
  tarifa = 0.25;
}
```

Este formato resulta ahora mucho más elegante y fácil de entender.

Todas las sentencias presentadas son variantes de una única sentencia IF de **C±** cuya sintaxis es la siguiente:

```
Sentencia_IF ::= if ( Condición ) { Secuencia_de_sentencias }
        { else if ( Condición ) { Secuencia_de_sentencias } }
        [ else { Secuencia_de_sentencias } ]
```

### 5.3.3 Sentencia WHILE

En **C±** la estructura de *iteración* de la figura 5.5 se consigue mediante la sentencia WHILE, que tiene el siguiente formato:

```
while ( Condición ) {
  Acción
}
```

El significado es que mientras la expresión *Condición* resulta cierta, se ejecuta la *Acción* de forma repetitiva. Cuando el resultado es falso finaliza la ejecución de la sentencia. Si la *Condición* resulta falsa en la primera evaluación, la *Acción* no se ejecuta nunca. Por ejemplo, el factorial de un número $n$ se puede calcular mediante la fórmula habitual:

$$n! = 1 \times 2 \times 3 \times 4 \times ... \times n$$

Este cálculo se puede programar en **C±** utilizando una sentencia `while` de la siguiente forma:

```
factorial = 1;
while (n > 1) {
   factorial = factorial * n;
   n--;
}
```

Así, con la sentencia de *autodecremento* la variable `n` va disminuyendo su valor de uno en uno en cada repetición del bucle, al tiempo que esos valores se van multiplicando sucesivamente, guardando el producto acumulado en **`factorial`**, hasta que `n` se reduce a 1. Si inicialmente el valor de `n` es igual o menor que 1, no se ejecutan nunca las sentencias dentro del bucle, por lo que la variable `factorial` termina con el mismo valor inicial igual a 1.

### 5.3.4 Sentencia FOR

Existen muchas situaciones en las que las repeticiones del bucle se controlan mediante una variable que va contando las veces que se ejecuta. La cuenta puede ser en sentido creciente, o decreciente. La *Condición* de la *iteración* se limita a comprobar si se ha alcanzado el límite correspondiente al número de repeticiones previstas. Esto es lo que hemos hecho en el ejemplo del factorial con la variable `n`. La variable `n` se decrementa en cada iteración y se comprueba el número total de ejecuciones con la expresión condicional `n > 1`.

Debido a lo habitual de esta situación, en casi todos los lenguajes existen sentencias que simplifican su construcción. En **C±** se dispone de la sentencia FOR, cuya forma para incremento creciente es la siguiente:

```
for (int Índice = Inicial ; Índice <= Final ; Índice ++) {
  Acción
}
```

El símbolo punto y coma (`;`) separa los distintos elementos de control de la sentencia. La actualización del índice se hace mediante la sentencia de *autoincremento*.

La variable *Índice* sirve de contador para controlar el número de iteraciones a realizar. Inicialmente la variable *Índice* toma el valor *Inicial* y se incrementa automáticamente en una unidad con cada nueva ejecución de *Acción*. La *Acción* se ejecuta repetidamente hasta que la variable *Índice* alcanza el valor *Final*.

Ambos valores, inicial y final, pueden ser expresiones aritméticas. Estas expresiones se evalúan sólo una vez al comienzo de la sentencia FOR y no se

modifican durante toda su ejecución. Si el valor inicial es mayor que el valor final, la *Acción* no se ejecuta nunca.

La variable *Índice* puede ser utilizada dentro de *Acción* pero nunca debe ser modificada, pues se perdería el control automático de las repeticiones. Esto constituye una norma del *Manual de Estilo* que es obligatorio aplicar a cualquier sentencia FOR.

La variable *Índice* se declara dentro del propio FOR, y sólo existe mientras se ejecuta. Al terminar la ejecución la variable *Índice* ya no es visible en las siguientes sentencias del programa.

Ahora se puede volver a escribir el cálculo del factorial de la siguiente manera:

```
factorial = 1;
for (int indice = 2; indice <= n; indice++) {
  factorial = factorial * indice;
}
```

Este fragmento de programa es más sencillo. Además, resulta evidente que para el cálculo del factorial se emplea un número de iteraciones conocido, que no depende de lo que se calcula con cada iteración. Como se puede observar dentro del bucle se utiliza la variable **indice** con el valor que toma en cada pasada y la variable n mantiene su valor inicial. Para valores de n inferiores a 2 las sentencias dentro del bucle no se ejecutan nunca.

La sentencia FOR de **C±** tiene una versión para decrementar el contador en cada repetición. En este caso el formato es el siguiente:

```
for (int Índice = Inicial ; Índice >= Final ; Índice --) {
  Acción
}
```

Ahora en cada iteración la variable *Índice* se decrementa en una unidad con cada nueva ejecución de *Acción* mediante la sentencia de *autodecremento*. Por ejemplo, se puede realizar el cálculo del factorial en sentido inverso de la manera siguiente:

```
factorial = 1;
for (int indice = n; indice <= 2; indice--) {
  factorial = factorial * indice;
}
```

La sintaxis completa de la sentencia FOR es, por tanto, la siguiente:

*Sentencia_FOR* **::=** *For_creciente* **|** *For_decreciente*

*For_creciente* **::=** `for ( int` *Variable_índice* = *Valor_inicial* ;
*Variable_índice* <= *Valor_final* ; *Variable_índice* ++ )
{ *Secuencia_de_sentencias* }

*For_decreciente* **::=** `for ( int` *Variable_índice* = *Valor_inicial* ;
*Variable_índice* >= *Valor_final* ; *Variable_índice* -- )
{ *Secuencia_de_sentencias* }

*Variable_índice* **::=** *Identificador*

*Valor_inicial* **::=** *Expresión*

*Valor_final* **::=** *Expresión*

## 5.4 Ejemplos de programas

En este apartado se muestran varios ejemplos de programas completos que utilizan las sentencias presentadas en este tema.

### 5.4.1 Ejemplo: Ordenar tres datos

Se trata de realizar un programa que lea 3 valores enteros y los ordene de menor a mayor en las mismas variables en que se leen: el valor menor quedará en la primera variable y el mayor en la última.

La ordenación se realiza en dos pasos: la primera parte se dedica a ordenar los dos primeros valores y en la segunda se ordenará el tercero comparándolo con los ya ordenados. Se utiliza una variable auxiliar para realizar los intercambios de valores entre las variables. El listado del programa completo esta recogido a continuación.

```
/*******************************************************
* Programa: Ordenar3
*
* Descripción:
*   Este programa ordena tres valores y
*   los guarda ordenados de menor a mayor
*   en las mismas variables que se leen
*******************************************************/
#include <stdio.h>
```

```
int main() {
  int valUno, valDos, valTres, auxiliar;

  /*-- Leer los datos --*/
  printf( "¿Datos? " );
  scanf( "%d %d %d", &valUno, &valDos, &valTres );

  /*-- Primer Paso: Ordenar los dos primeros datos --*/
  if (valUno > valDos) {
    auxiliar = valUno;
    valUno = valDos;
    valDos = auxiliar;
  }

  /*-- Segundo Paso: Situar el tercer dato --*/
  if (valTres < valUno) {
    auxiliar = valTres;
    valTres = valDos;
    valDos = valUno;
    valUno = auxiliar;
  } else if (valTres < valDos) {
    auxiliar = valDos;
    valDos = valTres;
    valTres = auxiliar;
  }

  /*-- Tercer Paso: Escribir el resultado --*/
  printf( "Datos Ordenados = %5d %5d %5d\n", valUno, valDos, valTres);
}
```

La ejecución del programa produce un resultado como el siguiente:

```
¿Datos? 567 123 89
Datos Ordenados =    89   123   567
```

### 5.4.2 Ejemplo: Escribir un triángulo de dígitos

Con este programa se trata de escribir un triángulo de números entre el 1 y el 9. El nivel del triángulo se introducirá como dato. El resultado del programa será semejante al mostrado a continuación:

```
¿Altura del triángulo? 9
                                1
                               121
                              12321
                             1234321
                            123454321
                           12345654321
                          1234567654321
                         123456787654321
                        12345678987654321
```

Este es un problema típico en el que se pueden utilizar un número de iteraciones conocidas:

- Primeramente se necesita una iteración por cada línea de dígitos. El número de líneas es un dato de partida que se introduce, por tanto, es conocido.
- Cada línea se puede realizar en dos mitades:
  - 1ª mitad: Una iteración creciente hasta el número de línea. El número de iteraciones es conocido e igual al número de la línea que se esta escribiendo. Por ejemplo para la sexta línea: `123456`
  - 2ª mitad: Una iteración decreciente desde el número de línea decrementada en uno, hasta uno. Por tanto, el número de iteraciones es conocido. Por ejemplo para la sexta línea: `54321`

La escritura del primer número se puede conseguir mediante otra iteración que escribe caracteres en blanco. Si el triángulo está centrado en `centro`, el primer número se tiene que escribir tantas posiciones delante del centro como el número de línea que se esta escribiendo. Hay que tener en cuenta que en cada iteración, la `altura` del triángulo coincide con el número de línea que se está escribiendo. El programa completo se muestra a continuación.

```
/*******************************************************
* Programa: TrianguloDeNumeros
*
* Descripción:
*   Este programa escribe un triángulo de dígitos.
*   La altura se lee como dato y debe ser menor de 10
*******************************************************/
#include <stdio.h>

int main() {
  const int centro = 35; /* Posición del eje del triángulo */
  const int inicial = 1; /* Dígito inicial: superior y laterales */
  int nivel;

  /*-- Leer los datos --*/ {
    printf( "¿Altura del triángulo? " );
    scanf( "%d", &nivel );
  }
  /*-- Una iteración por cada línea del triángulo --*/
  for ( int altura = inicial; altura <= nivel; altura++) {

    /*-- Paso 1º: Situar primer número de cada línea --*/ {
      for (int indice = 1; indice <= centro-altura; indice++) {
        printf( " " );
      }
      printf( "%d", inicial );
    }
    /*-- Paso 2º: Primera mitad de la línea del triángulo
         Escribir números consecutivos hasta altura --*/ {
      for (int indice = inicial+1; indice <= altura; indice++) {
        printf( "%1d",indice );
      }
    }
    /*-- Paso 3º: Segunda mitad de la línea del triángulo.
         Escribir números decrecientes hasta Inicial --*/ {
      for (int indice = altura-1; indice >= inicial; indice--) {
        printf( "%1d", indice );
      }
    }
    /*-- Paso 4º: Saltar a una nueva linea --*/ {
      printf( "\n" );
    }
  }
}
```

### 5.4.3 Ejemplo: Elaboración de tickets y resúmenes

Este programa es un ejemplo simplificado del programa de una máquina de expender tickets de entrada a un espectáculo. El precio del ticket dependerá de la edad del espectador, según se indicó en el ejemplo del apartado 5.3.2, agrupados en Niños, Jóvenes, Adultos y Jubilados. El programa acumulará las entradas que se expenden y la cantidad total recaudada.

El programa dispondrá de un menú general para seleccionar la elaboración del ticket, la elaboración del resumen o la finalización del programa. El programa seguirá elaborando tickets hasta que el taquillero dé por finalizada su tarea. En los resúmenes se mostrarán los datos acumulados de tickets expendidos y el total recaudado. El listado del programa es el siguiente:

```
/*****************************************************
* Programa: Ticket
*
* Descripción:
*   Este programa elabora el ticket de entrada y los
*   resúmenes de recaudación de un espectáculo.
*   El precio del ticket depende de la edad del
*   espectador (Niño, Joven, Adulto o Jubilado)
*****************************************************/
#include <stdio.h>

int main() {
  const float PrecioBase = 6.0; /* Precio de la butaca */
  int butacas = 0;              /* Número de butacas vendidas */
  int edad;                     /* Edad del cliente */
  float totalEuros = 0.0;       /* Total de euros recaudados */
  float precio;                 /* Precio de cada butaca */
  char opcion = ' ';            /* Opción del programa */
  char tecla = ' ';             /* Tecla Si/No */

  /*-- Bucle hasta opción F de fin --*/
  while (opcion != 'F') {
    printf( "Opción (Ticket, Resumen o Fin) ? " );
    opcion = ' ';
    while ((opcion != 'T') && (opcion != 'R') && (opcion != 'F')) {
      scanf( "%c", &opcion );
    }
```

```
    /*========================
       Elaboración del Ticket
     ========================*/
    if (opcion == 'T') {
      tecla = 'S';
      while (tecla == 'S') {
        printf( "Edad? " );
        scanf( "%d", &edad );
        butacas++;
        printf( ".-----------------------------.\n" );
        printf( "|       TICKET DE ENTRADA       |\n" );
        if (edad < 6) {            /* niño, gratis */
          printf( "|Gratis  " );
          precio = 0.0;
        } else if (edad < 18) {  /* joven, 50% */
          printf( "|Joven   " );
          precio = PrecioBase / 2.0;
        } else if (edad < 65) {  /* adulto, tarifa completa */
          printf( "|Adulto  " );
          precio = PrecioBase;
        } else {                   /* jubilado, 25% */
          printf( "|Jubilado" );
          precio = PrecioBase / 4.0;
        }
        totalEuros = totalEuros + precio;
        printf( "         Precio: %4.2f|\n", precio );
        printf( "'-----------------------------'\n\n" );
        printf( "Otro Ticket(S/N)? " );
        tecla = ' ';
        while ((tecla != 'S') && (tecla != 'N')) {
          scanf( "%c", &tecla );
        }
      }
    }
    /*==========================================
       Elaboración del Resumen de la recaudación
     ==========================================*/
    if (opcion == 'R') {
      printf( "    RESUMEN DE VENTAS \n" );
      printf( "    ---------------- \n" );
      printf( "%4d Butacas \n", butacas );
      printf( "Total Recaudado = %10.2f\n\n", totalEuros );
    }
  }
}
```

Una ejecución simplificada del programa da como resultado el siguiente:

```
Opción (Ticket, Resumen o Fin) ? T
Edad? 13
.-----------------------------.
|      TICKET DE ENTRADA      |
|Joven            Precio: 3.00|
'-----------------------------'

Otro Ticket(S/N)? S
Edad? 67
.-----------------------------.
|      TICKET DE ENTRADA      |
|Jubilado         Precio: 1.50|
'-----------------------------'

Otro Ticket(S/N)? N
Opción (Ticket, Resumen o Fin) ? R
    RESUMEN DE VENTAS
    -----------------
   2 Butacas
Total Recaudado =       4.50

Opción (Ticket, Resumen o Fin) ? F
```

# Tema 6

# Metodología de Desarrollo de Programas (II)

El objetivo de este tema es múltiple. En primer lugar se amplía la técnica de refinamientos sucesivos presentada en el tema 4 con la posibilidad de usar esquemas de selección e iteración.

En segundo lugar se introduce la idea de verificación formal de programas, aunque de una manera muy simplificada, en forma de razonamientos intuitivos sobre las estructuras utilizadas hasta el momento.

Finalmente se trata por primera vez el problema de la eficiencia, y se introduce de forma igualmente simplificada el concepto de complejidad algorítmica.

## 6.1 Desarrollo con esquemas de selección e iteración

En este apartado se ilustra el empleo de la técnica de refinamientos sucesivos explicada en el tema 4, ampliada ahora con la posibilidad de utilizar las nuevas estructuras de selección o iteración. Con ello se tienen tres posibilidades a la hora de refinar una acción compuesta:

- Organizarla como secuencia de acciones.
- Organizarla como selección entre acciones alternativas.
- Organizarla como iteración de acciones.

En el tema 4 se ha presentado ya la metodología para desarrollar esquemas de secuencia. A continuación se amplía la metodología con recomendaciones para desarrollar esquemas de selección e iteración.

### 6.1.1 Esquema de selección

Recordaremos que un *esquema de selección* consiste en plantear una acción compuesta como la realización de una acción entre varias posibles, dependiendo de ciertas condiciones. Es decir, se trata de elegir una sola entre varias posibles alternativas.

Para desarrollar un esquema de selección debemos identificar sus elementos componentes. Por tanto habrá que:

(a) Identificar cada una de las alternativas del esquema, y las acciones correspondientes.
(b) Identificar las condiciones para seleccionar una alternativa u otra.

Como ejemplo, aplicaremos estas recomendaciones al desarrollo de una acción compuesta para calcular cuántos días tiene el mes de Febrero de un cierto año. Reconoceremos que:

(a) Las alternativas son que tenga 28 días o que tenga 29. Las acciones serán asignar dicho valor a una variable que almacene el número de días.

```
dias = 28
```

o bien

```
dias = 29
```

(b) La condición para elegir una acción u otra es que el año sea bisiesto. De forma simplificada (pero válida para años entre 1901 y 2099) expresaremos la condición como equivalente a que el año sea múltiplo de cuatro.

```
anno % 4 == 0
```

Colocando cada elemento identificado en el lugar correspondiente del esquema, tendremos:

```
if (anno % 4 == 0) {
    dias = 29;
} else {
    dias = 28;
}
```

De manera similar se pueden desarrollar esquemas de selección simplificados, con sólo una acción condicionada, o esquemas de selección en cascada en que haya un número más o menos grande de alternativas. Por ejemplo, el esquema anterior podría replantearse realizando primero el tratamiento más común, es decir, que Febrero tenga 28 días, y luego corrigiendo este valor si es necesario.

```
dias = 28;
if (anno % 4 == 0) {
    dias = 29;
}
```

Como ejemplo de selección en cascada, desarrollaremos el cálculo de los días del mes, para cualquier mes del año. Las alternativas son:

| | |
|---|---|
| 31 días: | Enero, Marzo, Mayo, Julio, Agosto, Octubre, Diciembre. |
| 30 días: | Abril, Junio, Septiembre, Noviembre |
| 29 días: | Febrero (año bisiesto). |
| 28 días: | Febrero (año normal). |

Para simplificar las expresiones de condición, dejaremos para la última alternativa aquella en que la condición sea más compleja. En este caso sería la de los meses de 31 días, que son los más numerosos. Las otras alternativas podemos situarlas en un orden arbitrario. Al escribir las condiciones debemos tener en cuenta que si hay que evaluar una de ellas es porque todas las anteriores han resultado falsas:

```
if ((mes==2) && (anno % 4 == 0)) {
    dias = 29;
} else if (mes==2) {
    dias = 28;
} else if ((mes==4) || (mes==6) ||
           (mes==9) || (mes==11)) {
    dias = 30;
} else {
    dias = 31;
}
```

### 6.1.2 Esquema de iteración

Una iteración o bucle consiste en la repetición de una acción o grupo de acciones hasta conseguir el resultado deseado. Para desarrollar un *esquema de iteración* dentro de un programa deberemos identificar cada uno de sus elementos componentes. Al hacerlo hay que identificar simultáneamente las variables adecuadas para almacenar la información necesaria.

En líneas generales se podría proceder de la siguiente manera:

(a) Identificar las acciones útiles a repetir, y las variables necesarias. Precisar el significado de estas variables al comienzo y final de cada repetición.

(b) Identificar cómo actualizar la información al pasar de cada iteración a la siguiente. Puede ser necesario introducir nuevas variables.

(c) Identificar la condición de terminación. Puede ser necesario introducir nuevas variables e incluso acciones adicionales para mantenerlas actualizadas.
(d) Identificar los valores iniciales de las variables, y si es necesaria alguna acción para asignárselos antes de entrar en el bucle.

Además de los elementos anteriores, puede ser necesaria alguna acción adicional al comienzo o al final del bucle. Si son acciones muy simples, pueden considerarse parte del esquema de iteración como tal. Si son algo complejas será mejor considerarlas como acciones anteriores o posteriores, siguiendo un esquema secuencial.

El método explicado corresponde al caso general. En bastantes casos el desarrollo de un bucle es mucho más sencillo. Esto ocurre en particular cuando se programan bucles con contador, si las acciones a repetir pueden expresarse directamente en función del contador del bucle. En este caso los pasos b), c) y d) se refunden en uno sólo, consistente en determinar los valores inicial, final y el incremento del contador del bucle.

Desarrollaremos con la técnica indicada un fragmento de programa que imprima los términos de la serie de Fibonacci. Cada término de esta serie se obtiene sumando los dos anteriores. La serie comienza con los términos 0 y 1, que se suponen ya impresos antes del bucle. Se trata de calcular e imprimir tantos términos como sea posible.

Procediendo paso a paso, describiremos cada elemento del desarrollo de manera informal, seguido de su codificación en **C±**.

(a) *Acciones útiles a repetir:* Imprimir un término.

```
printf("%10d\n", termino);
```

*Variables necesarias:* El término a imprimir.

```
int termino;
```

*Valor al empezar la repetición:* Ultimo término impreso hasta el momento.

(b) *Actualizar las variables al pasar de una repetición a la siguiente:* Antes de imprimir, calcular el término actual a partir de los dos anteriores (se necesita tener almacenado el penúltimo término).

```
aux = termino + anterior;
anterior = termino;
termino = aux;
```

*Variables adicionales:* El penúltimo término, y una variable temporal.

```
int anterior;
int aux;
```

(c) *Condición de terminación:* El término siguiente excedería del rango de los enteros. Hay que evaluar la condición sin calcular explícitamente el valor de dicho término, porque se produciría "*overflow*".

```
INT_MAX-termino < anterior
```

(Obsérvese que esta expresión equivale en teoría a `INT_MAX < termino+anterior`)

(d) *Valores iniciales de las variables:* Los dos primeros términos, 0 y 1.

```
anterior = 0;
termino = 1;
```

El bucle completo sería:

```
int termino;
int anterior;
int aux;
. . . .
anterior = 0;
termino = 1;
while (INT_MAX-termino >= anterior) {
   aux = termino + anterior;
   anterior = termino;
   termino = aux;
   printf("%10d\n", termino);
}
```

## 6.2 Ejemplos de desarrollo con esquemas

A continuación se ilustra la técnica de desarrollo por refinamientos utilizando los esquemas de selección e iteración sobre algunos ejemplos típicos.

### 6.2.1 Ejemplo: Imprimir el contorno de un triángulo

Se trata de imprimir con asteriscos el perímetro de un triángulo aproximadamente equilátero, tal como se indica a continuación:

```
      *
     * *
    *   *
   *     *
  *       *
 *         *
* * * * * * *
```

Como parámetro del problema se especificará la altura del triángulo. Si la altura es $N$, habrá que imprimir $N$ líneas, cada una con la configuración de asteriscos que corresponda. El triángulo anterior tiene altura 7. Se pretende además que el triángulo aparezca ajustado a la izquierda; es decir, el primer asterisco de la línea inferior deberá empezar exactamente en la primera posición de esa línea. Las otras líneas empezarán con el espacio en blanco apropiado.

El planteamiento inicial del programa será:

*Imprimir el borde de un triángulo*

El primer paso de refinamiento consistirá en decidir si esta acción compuesta debe plantearse como una secuencia de acciones simples, o como selección entre alternativas, o como bucle. Observando que hay tres clases de líneas a imprimir, según el número de asteriscos que contienen, elegiremos la primera opción, y escribiremos:

*Imprimir el borde de un triángulo* ⟶
  *Imprimir el vértice superior*
  *Imprimir los bordes laterales*
  *Imprimir el borde inferior*

El orden en que deben realizarse las acciones viene determinado por el hecho de que la impresora va imprimiendo las líneas sucesivas de arriba a abajo.

A continuación refinaremos cada una de estas acciones. Para ello examinaremos con detalle los caracteres que hay que imprimir en cada línea, en función de la altura del triángulo (en particular, los espacios en blanco intermedios). Marcando con puntos los espacios en blanco, tendremos:

```
Línea 1   ......*         N-1 blancos
Línea 2   .....*.*        N-2 blancos,  1 blanco
Línea 3   ....*...*       N-3 blancos,  3 blancos
          ...*.....*
Línea k   ..*.......*     N-k blancos, 2k-3 blancos
          .*.........*
Línea N   *.*.*.*.*.*.*   N asteriscos espaciados
```

El refinamiento de la primera acción será en forma de secuencia:

*Imprimir el vértice superior* ⟶
  *Imprimir N-1 blancos*
  *Imprimir un asterisco*
  *Saltar a la línea siguiente*

La acción de imprimir los blancos será una iteración, que podemos escribir directamente mediante un bucle con contador:

*Imprimir N-1 blancos* ⟶

```
for (int k = 1; k <= N-1; k++) {
   printf( " " );
}
```

Las acciones de imprimir un asterisco y saltar a la línea siguiente se escriben inmediatamente:

*Imprimir un asterisco* ⟶

```
printf( "*" );
```

*Saltar a la línea siguiente* ⟶

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

En **C±** resulta más natural que estas acciones se agrupen en la misma sentencia:

*Imprimir un asterisco y Saltar a la línea siguiente* ⟶

```
printf( "*\n" );
```

En el siguiente paso de refinamiento detallaremos la impresión de los bordes laterales. Intuitivamente podemos establecer una estructura de iteración mediante un bucle con contador:

*Imprimir los bordes laterales* ⟶

```
for (int k = 2; k <= N-1; k++) {
   Imprimir los bordes de la línea k
}
```

Observaremos que la línea $k$-esima va precedida de $N - k$ blancos, y tiene $2k - 3$ blancos entre los bordes:

*Imprimir los bordes de la línea k* ⟶
  *Imprimir N-k blancos*
  *Imprimir un asterisco*
  *Imprimir 2k-3 blancos*
  *Imprimir un asterisco*
  *Saltar a la línea siguiente*

Todas estas acciones han sido ya refinadas como sentencias de **C±**. Finalmente falta por plantear la impresión del borde inferior. Podremos distinguir la impresión del primer asterisco y la del resto, y escribir:

*Imprimir el borde inferior* ⟶
*Imprimir un asterisco*
*Imprimir N-1 asteriscos precedidos de blanco*
*Saltar a la línea siguiente*

La acción central se escribirá como un bucle con contador:

*Imprimir N-1 asteriscos precedidos de blanco* ⟶

```
for (int k = 1; k <= N-1; k++) {
   printf( " *" );
}
```

Reuniendo todos los fragmentos, añadiendo la parte declarativa necesaria, y documentando cada elemento importante del programa, tendremos el programa completo, tal como se recoge en el listado de la versión inicial.

```
/*********************************************************
* Programa: Triangulo (Versión inicial)
*
* Descripción:
*   Este programa escribe el borde de un triángulo
*   aproximadamente equilátero, usando asteriscos.
*********************************************************/
#include <stdio.h>

int main() {
   const int N = 7;  /* altura del triángulo */

  /*-- Escribir el vértice superior --*/
  for (int k = 1; k <= N-1; k++) {
    printf( " " );
  }
  printf( "*\n" );

  /*-- Imprimir los bordes laterales --*/
  for (int k = 2; k <= N-1; k++) {
    for (int j = 1; j <= N-k; j++) {
      printf( " " );
    }
    printf( "*" );
```

```
    for (int j = 1; j <= 2*k-3; j++) {
      printf(" ");
    }
    printf( "*\n" );
  }

  /*-- Imprimir el borde inferior --*/
  printf( "*" );
  for (int k = 1; k <= N-1 ; k++) {
    printf( " *" );
  }
  printf( "\n" );
}
```

Este programa puede compilarse y ejecutarse, obteniendo el resultado puesto anteriormente como ejemplo. Sin embargo no puede considerarse totalmente terminado. Una deficiencia es que la altura del triángulo aparece como constante, cuando en realidad sería más razonable leerla como dato en cada ejecución del programa. La solución será definir `N` como variable, y leer su valor al comienzo.

Otra deficiencia es que no se tienen en cuenta algunos casos especiales, en que el triángulo degenera y no tiene todos los elementos identificados. Esto ocurre cuando la altura del triángulo es 0, 1 ó 2. En el primer caso se omiten todas las líneas; en el segundo sólo existe el vértice superior, y en el tercero no existen bordes laterales.

En esta versión inicial, para la altura igual a 2 no se obtienen resultados erróneos, ya que el bucle de líneas de los bordes laterales tal como está escrito se ejecutaría 0 veces (pero esto se debe más a la casualidad que a un razonamiento previo del comportamiento del programa). Por ejemplo, si cambiásemos el parámetro de altura al valor 1:

```
const N = 1;
```

obtendríamos el resultado erróneo:

```
*
*
```

este mismo resultado se obtiene para `N = 0` . La solución general será convertir en esquema de selección condicional la impresión del vértice superior y del borde inferior:

*Imprimir el vértice superior* ⟶

```
if (N>0) {
    Imprimir N-1 blancos
    Imprimir un asterisco
    Saltar a la línea siguiente
}
```

*Imprimir el borde inferior* ⟶

```
if (N>1) {
    Imprimir un asterisco
    Imprimir N-1 asteriscos precedidos de blanco
}
```

La versión corregida del programa aparece en el siguiente listado:

```
/********************************************************
* Programa: Triangulo2 (Versión corregida)
*
* Descripción:
*   Este programa escribe el borde de un triángulo
*   aproximadamente equilátero, usando asteriscos.
*   La altura del triángulo, en líneas de texto,
*   se lee como dato
********************************************************/
#include <stdio.h>

int main() {
  int altura;       /* altura del triángulo */

  /*-- Leer altura deseada --*/
  printf( "¿Altura? " );
  scanf( "%d", &altura );

  /*-- Imprimir el vértice superior --*/
  if (altura > 0) {
    for (int k = 1; k <= altura-1; k++) {
      printf( " " );
    }
    printf( "*\n" );
  }
```

```
  /*-- Imprimir los bordes laterales --*/
  for (int k = 2; k <= altura-1; k++) {
    for (int j = 1; j <= altura-k; j++) {
      printf( " " );
    }
    printf( "*" );
    for (int j = 1; j <= 2*k-3; j++) {
      printf( " " );
    }
    printf( "*\n" );
  }

  /*-- Imprimir el borde inferior --*/
  if (altura > 1) {
    printf( "*" );
    for (int k = 1; k <= altura-1 ; k++) {
      printf( " *" );
    }
    printf( "\n" );
  }
}
```

Algunos ejemplos de ejecución serian los siguientes:

```
¿Altura? 7
      *
     * *
    *   *
   *     *
  *       *
 *         *
* * * * * * *

¿Altura? 1
*
```

### 6.2.2 Ejemplo: Imprimir el triángulo de Floyd

Se trata de desarrollar un programa que imprima el llamado *triángulo de Floyd*. Este "triángulo" se forma imprimiendo los sucesivos números naturales 1, 2, 3, ... en filas sucesivas, colocando un número en la primera línea, dos en la segunda, tres en la tercera, etc. Si fijamos un límite arbitrario de la serie de números (p. ej. 12), tendremos el triángulo:

```
 1
 2   3
 4   5   6
 7   8   9  10
11  12
```

Es fácil darse cuenta de que en general la línea $k$ tiene $k$ números, excepto la última línea, que puede quedar incompleta. El programa de este ejemplo deberá leer como dato el límite de la serie.

El primer paso de refinamiento será:

*Imprimir el triángulo de Floyd* $\longrightarrow$
    *Leer el límite N de la serie*
    *Imprimir el triángulo hasta N*

La primera acción puede desarrollarse en sentencias de **C±** de forma inmediata:

*Leer el límite N de la serie* $\longrightarrow$

```
printf( "Límite de la serie: " );
scanf( "%d",&N );
```

La parte principal del programa es la impresión del triángulo. Habremos de refinarla en forma de un esquema de secuencia, selección o iteración. Puesto que se trata de imprimir un número variable de líneas y valores, con acciones similares para todos ellos, parece razonable usar un esquema de iteración. La decisión a tomar será si plantear la iteración como bucle de números o de líneas. Según la elección que hagamos tendremos dos esquemas de programa diferentes, que desarrollaremos por separado.

#### 6.2.2.1 Escritura mediante bucle de números

Elegiremos en primer lugar esta posibilidad, que facilita la codificación del bucle, ya que puede plantearse como un bucle con contador:

*Imprimir el triángulo hasta N* $\longrightarrow$

```
for (int k = 1; k <= N; k++) {
    Imprimir el número k
}
```

La impresión del número requiere acciones adicionales, ya que debe incluir el saltar de línea al comienzo de cada nueva línea del triángulo. Puede plantearse como secuencia de acciones, de la forma:

*Imprimir el número k* ⟶

```
|Saltar de línea, si es necesario
|printf( "%5d",k );
```

En este planteamiento los saltos de línea no se producen en cuanto se llega al final de la línea, sino cuando se va a escribir el siguiente valor. Esto quiere decir que el último número impreso dentro del bucle no irá seguido de salto de línea, y que habrá que completar la última línea como acción de terminación, fuera del bucle; es decir, la escritura del triángulo terminará con una acción adicional:

*Completar la última línea*

Refinaremos la acción de saltar de línea en forma de esquema de selección:

*Saltar de línea, si es necesario* ⟶

```
if (el número anterior fue el último de su línea) {
   printf( "\n" );
}
```

El refinamiento de la condición va a exigir introducir nuevos elementos. Para expresar dicha condición como una expresión en **C±** necesitaremos mantener actualizada la indicación de cuál es el último valor de cada línea. En todo caso necesitaremos conocer cuántos números han de imprimirse en la línea en curso (recordemos que en la línea $k$ habrá $k$ números).

Usaremos una variable `linea` como contador de líneas y otra variable `ultimo` para contener el último número de la línea. Estas variables habrán de actualizarse cada vez que se cambie de línea. Reescribiremos el refinamiento anterior de la forma siguiente:

*Saltar de línea, si es necesario* ⟶

```
if (k > ultimo) {
   printf( "\n" );
   linea++;
   ultimo = ultimo + linea;
}
```

Al mismo tiempo hay que ampliar la inicialización del bucle de escribir números incluyendo el dar valores iniciales a las nuevas variables, en concreto:

```
linea = 1;
ultimo = 1;
```

Esta inicialización es la adecuada incluso en el caso de escribir 0 números, ya que los valores corresponden al primer número que completa línea, tras el cual habrá que saltar de línea si se imprimen más números.

No hay que olvidar desarrollar la acción de saltar de línea al final de la última línea del triángulo. Esta acción será condicional si hemos de considerar el caso de que se manden escribir 0 números, en cuyo caso no existe esa última línea.

*Completar la última línea* ⟶

```
if (N > 0) {
  printf( "\n" );
}
```

Con todo ello podemos ya redactar el programa `Floyd1` completo, tal como aparece a continuación, documentándolo de la manera habitual.

```
/********************************************************
* Programa: Floyd1
*
* Descripción:
*   Este programa imprime el triángulo de Floyd
*   con los números correlativos de 1 a N
*   El valor de N se lee como dato
********************************************************/
#include <stdio.h>

int main() {
  int N;               /* último número a imprimir */
  int linea = 1;       /* contador de líneas */
  int ultimo = 1;      /* último número de la línea */

  /*-- Leer límite de la serie --*/
  printf( "¿Límite de la serie? " );
  scanf( "%d", &N );

  /*-- Imprimir el triángulo mediante un bucle de números --*/
  for (int k = 1; k <= N; k++) {
    /* saltar de línea, si es necesario --*/
    if (k > ultimo) {
      printf( "\n" );
      linea++;
      ultimo = ultimo+linea;
    }

    /* imprimir el número k */
    printf( "%5d", k );
  }
```

```
  /*-- Acabar la última línea --*/
  if (N > 0) {
    printf( "\n" );
  }
}
```

Un ejemplo de la ejecución del programa sería la siguiente:

```
¿Límite de la serie? 12
   1
   2   3
   4   5   6
   7   8   9  10
  11  12
```

#### 6.2.2.2 Escritura mediante bucle de líneas

La segunda alternativa de diseño consiste en plantear la escritura del triángulo como un bucle de imprimir líneas de números, una tras otra. El refinamiento correspondiente será:

*Imprimir el triángulo hasta N* ⟶

> **while** (*quedan números*) {
>     *Imprimir la siguiente línea*
> }

Para ir imprimiendo cada línea necesitaremos conocer el rango de números que le corresponde a cada línea. Usaremos unas variables similares a las de la solución anterior para mantener el número de la línea, y el primer y último número de esa línea. Al comienzo del bucle estas variables tendrán los valores correspondientes a la última línea impresa. Así, los valores iniciales serán:

```
int linea = 0;
int primero = 0;
int ultimo = 0;
```

Ahora podemos refinar los elementos del bucle:

*quedan números* ⟶

> `ultimo < N`

*Imprimir la siguiente línea* ⟶

> *Actualizar los límites*
> *Imprimir los números*

*Actualizar los límites* ⟶

```
linea++;
primero = ultimo + 1;
ultimo = ultimo + linea;
```

*Imprimir los números* ⟶

```
for (int k = primero; k <= ultimo; k++) {
   printf( "%5d",k );
}
printf( "\n" );
```

Con esto se completa el programa; pero antes de presentarlo en su conjunto es preciso realizar alguna corrección. Dicha corrección es debida a que al calcular los límites de una línea de números no se ha tenido en cuenta que la última línea puede no estar completa. El último número de la línea no debe ser nunca mayor que el último de la serie completa. Escribiremos, por tanto:

*Actualizar los límites* ⟶

```
linea++;
primero = ultimo + 1;
ultimo = ultimo + linea;
if (ultimo > N) {
   ultimo = N;
}
```

Ahora sí se puede construir el programa **Floyd2**, y presentarlo debidamente documentado tal como se hace en el siguiente listado. El resultado de la ejecución de este programa es idéntico al de la variante anterior. Ambos programas son equivalentes desde el punto de vista de su utilización.

```
/*******************************************************
* Programa: Floyd2
*
* Descripción:
*   Este programa imprime el triángulo de Floyd
*   con los números correlativos de 1 a N
*   El valor de N se lee como dato
********************************************************/
#include <stdio.h>

int main() {
  int N;               /* último número a imprimir */
  int linea = 0;       /* contador de líneas */
  int primero = 0;     /* primer número de la línea */
  int ultimo = 0;      /* último número de la línea */
```

```
  /*-- Leer límite de la serie --*/
  printf( "¿Limite de la serie? " );
  scanf( "%d", &N );

  /*-- Imprimir el triángulo mediante un bucle de líneas --*/
  while (ultimo < N) {
    /*-- actualizar los límites --*/
    linea++;
    primero = ultimo+1;
    ultimo = ultimo+linea;
    if (ultimo > N) {
      ultimo = N;
    }

    /*-- imprimir los números --*/
    for (int k = primero; k <= ultimo; k++) {
      printf( "%5d", k );
    }
    printf( "\n" );
  }
}
```

## 6.3 Verificación de programas

Ya se ha indicado que uno de los objetivos de la programación es la *corrección*. Un programa es correcto si produce siempre resultados de acuerdo con la especificación del programa. Evidentemente sólo tiene sentido hablar de corrección si antes de escribir el programa se ha escrito de manera precisa la especificación del comportamiento que se espera que tenga.

En la práctica, la verificación de un programa se hace muchas veces mediante ensayos. Un *ensayo* (en inglés, *testing*) consiste en ejecutar el programa con unos datos preparados de antemano y para los cuales se sabe cuál ha de ser el resultado a obtener. Si al ejecutar el programa no se obtienen los resultados esperados, se sabrá que hay algún error, y el programa se examina para determinar la causa del error y eliminarla. Este proceso se llama *depuración* (en inglés *debugging*).

Si la ejecución produce los resultados esperados, entonces el ensayo no suministra ninguna información acerca de la corrección del programa. Puede ser que el programa sea correcto, y no tenga errores, pero también puede ocurrir

que el programa contenga errores que sólo se pongan de manifiesto con otros datos de ensayo diferentes.

La única manera de verificar con seguridad la corrección de un programa es demostrar formalmente que el programa cumple con sus especificaciones. Para ello es necesario escribir esas especificaciones con toda precisión en forma de expresiones lógicas, y luego realizar una demostración lógico-matemática de que el programa las cumple. A continuación se exponen algunas ideas sobre cómo puede ser este proceso de demostración.

De todas formas hay que comprender que la demostración formal de la corrección de un programa no significa necesariamente que el programa cumpla con sus objetivos reales, ni que el resultado sea correcto desde el punto de vista del usuario. Las técnicas de demostración formal que se introducen en este tema sólo sirven para demostrar que el programa cumple con sus especificaciones. El que dichas especificaciones describan realmente el problema a resolver es una cuestión aparte, mucho más difícil de asegurar, y que desde luego no admite una demostración lógico-matemática.

No obstante, los métodos formales de especificación, desarrollo y prueba de programas han mostrado ser mucho más seguros que los basados en la intuición de los desarrolladores o bien en ensayos con datos de prueba particulares.

### 6.3.1 Notación lógico-matemática

Las especificaciones formales utilizadas en este texto se basan en una notación lógico-matemática convencional, con ciertas adaptaciones. Por ejemplo, se usa la notación de vectores de **C±** en lugar de subíndices, y se generalizan los operadores sobre conjuntos para usarlos también con otros tipos de colecciones.

En el apéndice C se incluye una descripción detallada de la notación empleada, con sus particularidades.

### 6.3.2 Corrección parcial y total

Usando las expresiones y reglas de la lógica, y conociendo la semántica (significado) de las acciones, es posible demostrar si un programa es o no correcto (respecto a una determinada especificación). Para programas que siguen el modelo imperativo el proceso de demostración se realiza en dos partes:

1. *Corrección parcial* : si el programa termina el resultado es correcto.
2. *Corrección total*: lo anterior, y además para todo dato de entrada válido el programa termina

La base de la comprobación de corrección parcial es:

- Anotar el comienzo y final del programa con *aserciones* o asertos (afirmaciones, formalizadas como expresiones lógicas) correspondientes a las condiciones iniciales y al resultado deseado. La condición al comienzo se suele denominar *precondición*, y la del final *postcondición*. La precondición y la postcondición, conjuntamente, constituyen la *especificación formal* del programa.
- Anotar los puntos intermedios del programa con aserciones similares respecto al estado del cómputo en ese punto.
- Demostrar que si se cumple una aserción en un punto del programa y se siguen cada una de las líneas de ejecución posibles hasta llegar a otro punto con aserción, dicha aserción ha de cumplirse, según las reglas de la lógica y de acuerdo con las acciones realizadas.

En particular, la corrección parcial se consigue al demostrar que al llegar al final del programa ha de cumplirse la aserción correspondiente al resultado deseado.

La corrección total se consigue añadiendo la demostración de que todos los bucles del programa terminan tras un número finito de repeticiones. Para demostrar la terminación se puede:

- Asociar a cada bucle una función monótona (siempre estrictamente creciente o decreciente) llamada *variante*, y que debe tener un valor acotado para que el bucle se repita.

De esta manera tras un cierto número de repeticiones se alcanzará la cota o límite de dicha función, y el bucle terminará.

Existen diferentes notaciones para escribir las aserciones que se pueden asociar a cada punto importante del programa. En los ejemplos que siguen se usarán expresiones lógico-matemáticas convencionales, encerradas entre comillas tipográficas «...» para distinguirlas del código del programa en **C±**. La notación se describe en el apéndice C.

### 6.3.3 Razonamiento sobre sentencias de asignación

Para analizar el comportamiento de un fragmento de programa correspondiente a una sentencia de asignación, comenzaremos por anotar delante de dicha sentencia todas las condiciones que sabemos que se cumplen inmediatamente antes de ejecutarla. A continuación anotaremos detrás de la sentencia las condiciones que podamos demostrar que se cumplen después de su ejecución, y que serán las siguientes:

- Las condiciones anteriores en las que no intervenga la variable asignada.
- La condición de que la variable tiene el valor asignado.

Por ejemplo (asumiendo que es cierto lo que se indica al comienzo de esta sentencia):

«$ASERTO$: $(x > y) \wedge (a > b) \wedge (a > x)$»
```
a = 36;
```
«$ASERTO$: $(x > y) \wedge (a = 36)$»

En la anotación final se han suprimido las condiciones iniciales $(a > b)$ y $(a > x)$, ya que en ellas interviene la variable $a$. Se ha conservado la condición $(x > y)$, ya que estas variables no son afectadas por la asignación, y se ha añadido la nueva condición $(a = 36)$ impuesta por la ejecución de la sentencia de asignación.

### 6.3.4 Razonamiento sobre el esquema de selección

Para analizar el comportamiento de un fragmento de programa correspondiente a un esquema de selección, comenzaremos por anotar delante de dicho esquema las condiciones que sepamos que se cumplen inmediatamente antes de examinar la condición.

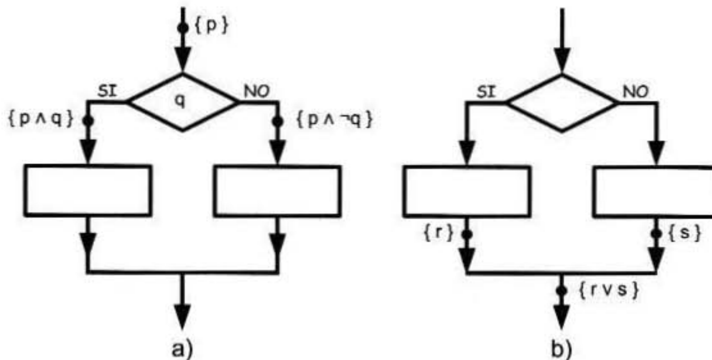



Figura 6.1 Razonamiento sobre un esquema de selección.

Puesto que la condición de selección decide la continuación por una u otra vía de las dos posibles, deduciremos que al comienzo de la alternativa "Sí" se cumplirán las condiciones iniciales y además la condición de selección, y que al comienzo de la alternativa "No" se cumplirán las condiciones iniciales y no se cumplirá la condición de selección. La figura 6.1(a) refleja gráficamente estas anotaciones.

En la parte de terminación del esquema anotaremos las condiciones que se deduzcan de la ejecución de cada alternativa en particular; y anotaremos como condición a la salida que ha de cumplirse alguna de las condiciones de terminación, correspondientes a las dos alternativas posibles, tal como se indica en la figura 6.1(b).

Aplicaremos esta técnica de razonamiento a un fragmento de programa que calcule en $m$ el máximo de dos números $a$ y $b$. Dicho fragmento podría ser:

```
if (a > b) {
  m = a;
} else {
  m = b;
}
```

Anotando las aserciones al comienzo, tendremos:

```
if (a > b) {
  «ASERTO: a > b»
  m = a;
} else {
  «ASERTO: a ≤ b»
  m = b;
}
```

Razonando sobre cada sentencia de asignación, de acuerdo con las propiedades matemáticas del máximo de dos valores, tendremos:

```
if (a > b) {
  «ASERTO: a > b»
  m = a;
  «ASERTO: (a > b) ∧ (m = a)» ⇒ «ASERTO: m = Max(a, b)»
} else {
  «ASERTO: a ≤ b»
  m = b;
  «ASERTO: (a ≤ b) ∧ (m = b)» ⇒ «ASERTO: m = Max(a, b)»
}
```

Se ha utilizado el símbolo $\Rightarrow$ para indicar que la segunda aserción se deduce de la primera, aplicando las propiedades de las operaciones lógico-matemáticas. Ahora se puede escribir la aserción final como unión de las dos alternativas (que en este caso coinciden):

```
if (a > b) {
  «ASERTO: a > b»
  m = a;
  «ASERTO: (a > b) ∧ (m = a)» ⇒ «ASERTO: m = Max(a, b)»
```

```
} else {
  «ASERTO: a ≤ b»
  m = b;
  «ASERTO: (a ≤ b) ∧ (m = b)» ⇒ «ASERTO: m = Max(a, b)»
}
«ASERTO: m = Max(a, b) ∨ m = Max(a, b)» ⇒ «ASERTO: m = Max(a, b)»
```

Con lo que queda demostrado el funcionamiento correcto.

### 6.3.5 Razonamiento sobre el esquema de iteración: invariante, terminación.

Para analizar el comportamiento de un fragmento de programa correspondiente a un esquema de iteración habremos de identificar, por una parte, las condiciones que deben cumplirse siempre inmediatamente antes de examinar la condición de repetición. Estas condiciones constituyen el llamado *invariante* del bucle. En la figura 6.2 se representa el diagrama de flujo de un bucle tipo WHILE, en que el invariante es la condición «$p$».

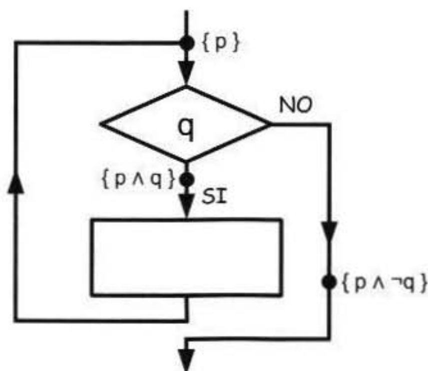



Figura 6.2 Razonamiento sobre un esquema de iteración.

Razonando como en el esquema de selección, deduciremos que al comienzo de cada repetición de la acción del bucle habrá de cumplirse el invariante y además la condición de repetición; y que al terminar las repeticiones y salir del bucle se cumplirá el invariante y además no se cumplirá la condición de repetición.

La identificación del invariante puede ser complicada. No se trata simplemente de anotar todas las condiciones que sabemos que se cumplen al llegar al bucle por primera vez, sino precisamente aquellas que además se seguirán

cumpliendo después de cada repetición. La denominación de *invariante* responde al hecho de que estas condiciones no cambian al ejecutar la acción del bucle.

Por otra parte, para garantizar la terminación hay que identificar una función estrictamente monótona y acotada que vaya cambiando con cada repetición del bucle. Al ser acotada no podrá ir variando indefinidamente, y eso permite asegurar que el bucle terminará en un número finito de repeticiones. A esta función se le denomina *variante*. Sin pérdida de generalidad se puede plantear que esta función debe tomar valores positivos enteros decrecientes, con el valor cero como cota inferior.

Como ejemplo, analizaremos un fragmento de programa para calcular en $f$ el factorial de un número $n$ ($n! = 1 \times 2 \times 3 \times ... \times n$). Para ello se usará un contador $k$ que vaya tomando los valores de 1 a $n$:

```
k = 1;
f = 1;
while (k < n) {
  k++;
  f = f * k;
}
```

Usaremos como invariante «$(k \leq n) \wedge (f = k!)$». Esto es válido para los casos en que $(n \geq 1)$. Como variante podremos usar la expresión $n - k$. Las anotaciones en el bucle serán:

```
k = 1;
f = 1;
«INVARIANTE : (k ≤ n) ∧ (f = k!)»  «VARIANTE : n − k»
while (k < n) {
  «ASERTO : (k < n) ∧ (f = k!)»
  k++;
  «ASERTO : (k ≤ n) ∧ (f = (k − 1)!)»
  f = f * k;
  «ASERTO : (k ≤ n) ∧ (f = k!)»
}
«ASERTO : (k ≤ n) ∧ (k ≥ n) ∧ (f = k!)» ⇒ «ASERTO : f = n!»
```

La localización exacta del invariante (ver figura 6.2) no corresponde a un punto claro en el código. En este ejemplo se ha escrito delante del bucle, marcándola expresamente con el nombre INVARIANTE. Lamentablemente al ponerla en ese lugar no queda del todo claro que debe cumplirse no sólo al entrar en el bucle por primera vez, sino también tras cada ejecución del bucle. Quizá un punto apropiado para esa anotación sería entre la palabra clave `while` y

la expresión de la condición (k < n), pero esto también puede resultar algo confuso:

```
while «(k ≤ n) ∧ (f = k!)» (k < n) {
  ...
}
```

La demostración de las anotaciones es relativamente sencilla, conociendo propiedades matemáticas tales como:

$$\begin{array}{l} 1! = 1 \\ (k < n) \rightarrow (k+1 \leq n) \\ k! = (k-1)! \times k \\ (k \leq n) \wedge (k \geq n) \rightarrow k = n \end{array}$$

El análisis anterior sólo es válido si $(n \geq 1)$. Falta comprobar la corrección para el caso $(n = 0)$, ya que para $(n < 0)$ el factorial no está definido. Sabiendo que $0! = 1$, y comprobando que si $(n = 0)$ el bucle no se ejecuta nunca, y la variable $f$ conserva su valor inicial $f = 1 = 0!$ tenemos la demostración completa.

Los razonamientos anteriores corresponden sólo a la demostración de corrección parcial. Para demostrar la corrección total falta demostrar que la expresión variante debe tener valor positivo para que el bucle se repita, y que decrece estrictamente con cada repetición. Y efectivamente, la expresión $n-k$ debe ser mayor que cero para que se cumpla la condición de repetición (k < n), y además disminuye cada vez que se incrementa $k$ dentro del bucle con la sentencia k++.

## 6.4 Eficiencia de programas. Complejidad

Ya se ha dicho que el objetivo prioritario de la programación es la corrección. La eficiencia sólo debe tenerse en cuenta si es un factor decisivo o importante en cada caso. Aunque en la primera redacción de un programa no conviene prestar excesiva atención a los aspectos de eficiencia, tampoco debe descuidarse totalmente este aspecto. Por esta razón se presentan a continuación los elementos básicos de la eficiencia de programas, y una descripción informal de algunas técnicas para analizar dicha eficiencia.

### 6.4.1 Medidas de eficiencia

La *eficiencia* de un programa se define en función de la cantidad de recursos que consume durante su ejecución. Un programa eficiente consume pocos recursos,

mientras que un programa menos eficiente consumirá una mayor cantidad de recursos. Esto quiere decir que establecer una medida de la eficiencia de los programas equivale a establecer una medida de los recursos usados durante su ejecución.

Las principales medidas de recursos empleados son:

- El tiempo que tarda en ejecutarse un programa.
- La cantidad de memoria usada para almacenar datos.

En muchos casos estos dos factores son mutuamente dependientes. Es decir, se pueden desarrollar programas que obtengan los resultados en menos tiempo, a costa de usar una mayor cantidad de memoria para almacenar datos, y viceversa.

En lo que sigue atenderemos exclusivamente a la primera de las dos medidas mencionadas. Un programa se considerará tanto más eficiente cuanto menos tiempo tarde en ejecutarse. Los programas poco eficientes tardarán mucho tiempo en dar los resultados. Hablaremos, por tanto, de la *eficiencia en tiempo* de un programa.

El tiempo de ejecución de un programa depende, en la mayoría de los casos, de los datos particulares con los que opera. Esto quiere decir que la eficiencia de un programa debe establecerse no como una magnitud fija para cada programa, sino como una función que nos dé el tiempo de ejecución para cada tamaño o cantidad de los datos que deba procesar.

Esta idea nos lleva, por su parte, a la necesidad de establecer previamente una medida del tamaño de los datos o *tamaño del problema*, para, en función de ella, establecer la medida de la eficiencia del programa que los procesa.

El tamaño del problema se puede expresar bien por la cantidad de datos a tratar, o bien por los valores particulares de los datos. Por ejemplo, para un programa que obtenga el valor medio o la suma de una serie de datos, el número de datos a sumar o promediar es una buena medida del tamaño del problema. En cambio, para un programa que calcule una potencia de un número, el tamaño significativo puede ser el valor del exponente.

La función que da el tiempo de ejecución según el tamaño del problema se dice que mide la *complejidad algorítmica* del programa.

### 6.4.2 Análisis de programas

La determinación de la eficiencia (o complejidad) de un programa se hace analizando los siguientes elementos:

1. Cuánto tarda en ejecutarse cada instrucción básica del lenguaje utilizado.
2. Cuántas instrucciones de cada clase se realizan durante una ejecución del programa

Para simplificar, consideraremos que cada operación elemental del lenguaje de programación: suma, resta, lectura, escritura, asignación de valor, decisión según condición, etc..., dura una unidad de tiempo. Con esta simplificación, el análisis de la eficiencia de un programa se centra en establecer cuántas instrucciones se ejecutan en total, dependiendo del tamaño o cantidad de los datos a procesar.

Al realizar el análisis mencionado nos encontraremos con que el número preciso de instrucciones ejecutadas depende de los valores particulares de los datos, incluso para un tamaño fijo del problema. En este caso se pueden adoptar al menos dos criterios diferentes para realizar el análisis de eficiencia:

- Analizar el comportamiento, en promedio.
- Analizar el comportamiento en el peor caso.

Utilizaremos el segundo criterio, por ser el más sencillo de aplicar, para analizar algunos ejemplos de programas. Con este criterio el análisis de complejidad (número de instrucciones ejecutadas) de los esquemas básicos de los programas se basa en las siguientes reglas:

1. La complejidad de un esquema de secuencia es la suma de las complejidades de sus acciones componentes.
2. La complejidad de un esquema de selección equivale a la de la alternativa más compleja, es decir, de ejecución más larga, más la complejidad de la evaluación de la condición de selección.
3. La complejidad de un esquema de iteración se obtiene sumando la serie correspondiente al número de instrucciones en las repeticiones sucesivas.

Veamos cómo es este análisis en algunos casos concretos. Tomemos como ejemplo el siguiente fragmento de programa que obtiene el máximo de dos números:

```
maximo = a;
if (a < b) {
  maximo = b;
}
```

El esquema global es una secuencia de dos acciones: una asignación, seguida de un esquema de selección (`if`). Anotaremos el programa con el número de instrucciones correspondientes a cada sentencia, para lo cual contaremos el número de operadores (`+`, `-`, `*`, `<`, `=`, etc.) y decisiones (`if`, `else`, `while`, etc.). Aplicando las reglas para los esquemas tendremos:

| Código | Número de instrucciones ejecutadas | | |
|---|---|---|---|
| `maximo = a;` | | 1 | |
| `if (a < b) {` | 2 | | |
| `  maximo = b;` | 1 | 3 | (Regla 2) |
| `}` | | | |
| | Total = | 4 | (Regla 1) |

La complejidad en este caso es fija, y no depende de una medida de tamaño del problema.

A continuación analizaremos de manera similar un bucle que obtiene en $f$ el factorial de un número $n$. Anotaremos el programa con el número de instrucciones de cada sentencia:

| Código | Número de instrucciones ejecutadas | | |
|---|---|---|---|
| `k = 1;` | | 1 | |
| `f = 1;` | | 1 | |
| `while (k < n) {` | 2 | | |
| `  k++;` | 1 | 5(n-1) | (Regla 3) |
| `  f = f * k;` | 2 | | |
| `}` | | | |
| | Total = | 5n - 3 | (Regla 1) |

Para calcular el número de instrucciones del bucle se ha multiplicado el número de instrucciones en cada repetición por el número de repeticiones. La complejidad aparece expresada en función del valor de $n$, que en este caso resulta una medida natural del tamaño del problema.

### 6.4.3 Crecimiento asintótico

En los análisis de eficiencia (o complejidad) se considera muy importante la manera como la función de complejidad va aumentando con el tamaño del problema. Lo que interesa es la forma de crecimiento del tiempo de ejecución, y no tanto el tiempo particular empleado.

Como ejemplo podemos comparar dos programas, uno que tarde un tiempo $100N$ en resolver un problema de tamaño $N$, y otro que tarde un tiempo $N^2$. La comparación puede hacerse escribiendo una tabla con los tiempos de cada uno para diferentes tamaños del problema.

Al principio de la tabla el primer programa parece menos eficiente que el segundo, ya que tarda mucho más tiempo, pero a medida que aumenta el tamaño del problema ambos programas llegan a tardar lo mismo (para tamaño 100),

y a partir de ahí el segundo programa demuestra ser mucho menos eficiente que el primero.

| **Tamaño** | **100$N$** | $N^2$ |
|---:|---:|---:|
| 1 | 100 | 1 |
| 2 | 200 | 4 |
| 3 | 300 | 9 |
| 10 | 1.000 | 100 |
| 100 | 10.000 | 10.000 |
| 1000 | 100.000 | 1.000.000 |

La menor eficiencia del segundo programa para tamaños grandes del problema no cambia por el hecho de que se modifique el coeficiente multiplicador. Si el primer programa tardase 10 veces más ($1000N$ en lugar de $100N$), acabaría igualmente por resultar mejor que el segundo a partir de un cierto tamaño del problema.

Lo que importa es la forma de la función, que en el primer caso es *lineal*, y en el segundo es *cuadrática*. La forma en que crece la función para tamaños grandes se dice que es su *comportamiento asintótico*, y se representa mediante la notación:

$$\mathbf{O}(f(n))$$

En dicha notación $n$ indica el tamaño del problema, $f$ la forma o función de crecimiento asintótico, y $\mathbf{O}$ (que se lee O-grande) significa orden de crecimiento. Algunas formas de crecimiento típicas, y sus valoraciones habituales, son las siguientes:

| | |
|---|---|
| $\mathbf{O}(1)$ | Complejidad constante. Ideal. |
| $\mathbf{O}(\log n)$ | Complejidad logarítmica. Muy eficiente. |
| $\mathbf{O}(n)$ | Complejidad lineal. Muy eficiente. |
| $\mathbf{O}(n \log n)$ | Complejidad lineal-logarítmica. Eficiente. |
| $\mathbf{O}(n^2)$ | Complejidad cuadrática. Menos eficiente. |
| $\mathbf{O}(n^k)$ | Complejidad polinómica. Poco eficiente. |
| $\mathbf{O}(2^n)$ | Complejidad exponencial. Muy poco eficiente. |

Las valoraciones indicadas son, por supuesto, muy imprecisas. Hay muchos problemas que sólo pueden resolverse mediante programas de complejidad elevada, poco eficientes según esta valoración, pero adecuados para esos problemas. Los problemas que sólo pueden resolverse con programas de complejidad exponencial se consideran *problemas intratables* en la práctica para tamaños grandes.

# Ejercicios sin resolver - I

Hasta este momento se han presentados los conceptos básicos de programación con diversos ejemplos completamente resueltos. Es aconsejable que estos ejercicios ya resueltos sólo sean consultados como procedimiento de autocorrección, después de intentar una solución propia. En todo caso, a continuación se enuncian varios ejercicios sin resolver semejantes a los ya resueltos. Todos ellos se pueden realizar utilizando las herramientas y metodología ya explicadas. Es muy importante que la realización de los ejercicios se lleve a cabo en dos etapas:

1. Plantear sobre el papel la solución del ejercicio, empleando la técnica de refinamientos sucesivos ya explicada.
2. Comprobar en el computador la solución adoptada.

Así como no es necesario buscar una solución propia a todos los ejercicios resueltos, tampoco es necesario resolver todos estos ejercicios propuestos. El objetivo de todos ellos es facilitar el aprendizaje de una buena metodología de programación.

Los enunciados de los ejercicios son los siguientes:

1. Realizar un programa que imprima la tabla de multiplicar por un número leído como dato. Por ejemplo, si se quiere obtener la tabla de multiplicar del 9 se tendría la siguiente hoja de resultados:

```
Numero? 9

Tabla de multiplicar por 9
==========================
     9  x  1  =   9
     9  x  2  =  18
     9  x  3  =  27
     9  x  4  =  36
     9  x  5  =  45
     9  x  6  =  54
     9  x  7  =  63
     9  x  8  =  72
     9  x  9  =  81
     9  x 10  =  90
```

2. Realizar un programa para calcular el máximo común divisor de dos números enteros. Por ejemplo:

```
Primer Numero? 655
Segundo Número? 1325

El máximo común divisor es:     5
```

3. Realizar un programa que escriba un rombo simétrico de asteriscos como el que se muestra a continuación, tomando como dato el número de asteriscos que tiene el lado.

```
Lado? 4

   *
  * *
 * * *
* * * *
 * * *
  * *
   *
```

4. Realizar un programa que calcule el número $e$ mediante el desarrollo en serie:

$$e=1+\sum_{i=1}^{n}\frac{1}{i!}$$

con un error menor del introducido como dato. Por ejemplo:

```
Error ? 0.000001

Valor de e = 2.71828198
```

5. Realizar un programa que lea la longitud de los tres lados de un triángulo y analice qué tipo de triángulo es. Los resultados posibles serán los siguientes:
   - No forman triángulo (un lado mayor que la suma de los otros dos).
   - Triángulo equilátero (tres lados iguales).
   - Triángulo isósceles (dos lados iguales).
   - Triángulo escaleno (tres lados distintos).
   - Triángulo rectángulo (sus lados cumplen el teorema de Pitágoras).
6. Realizar un programa que analice un texto terminado con un punto (`.`) y contabilice los siguientes aspectos:
   - Número total de caracteres.
   - Número total de vocales utilizadas.
   - Total de veces utilizada la vocal "a" mayúscula o minúscula.
   - Total de veces utilizada la vocal "e" mayúscula o minúscula.
   - Total de veces utilizada la vocal "i" mayúscula o minúscula.
   - Total de veces utilizada la vocal "o" mayúscula o minúscula.
   - Total de veces utilizada la vocal "u" mayúscula o minúscula.
7. Realizar un programa que dado un número $N$, introducido como dato, escriba todos los números comprendidos entre 1 y 10000 que cumplan las dos reglas siguientes:
   - Regla 1: La suma de sus cifras debe ser un divisor de $N$.
   - Regla 2: El producto de sus cifras debe ser un múltiplo de $N$.
8. Realizar un programa que a partir del capital ($C$), el tanto por ciento de interés anual ($I$) y los años de amortización ($A$) de un crédito, introducidos como datos, calcule la anualidad fija a pagar a lo largo de los $A$ años. La formula para este cálculo es la siguiente:

$$Anualidad = C\frac{\left(1+\frac{I}{100}\right)^{A}\frac{I}{100}}{\left(1+\frac{I}{100}\right)^{A}-1}$$

El programa también debe calcular para todos los años la parte de la anualidad dedicada al pago de intereses y la parte dedicada a la amortización de la deuda. Por ejemplo:

```
Capital? 1000000
Interés (%)? 15
Años? 3

Anualidad: 437977

Año          Intereses          Amortización
 1            150000               287977
 2            106804               331173
 3             57127               380850
```

# Tema 7

# Funciones y Procedimientos

El concepto de subprograma es fundamental para poder desarrollar programas grandes. Este tema y el siguiente se dedican por entero a introducir dicho concepto.

Las dos formas clásicas de subprogramas, disponibles prácticamente en cualquier lenguaje imperativo, son las funciones y los procedimientos. En este tema se detalla cómo se definen y utilizan ambos tipos de subprogramas y las diferencias que existen entre ellos.

El tema se complementa haciendo explícitas las principales dificultades de uso que conlleva el empleo de subprogramas para los programadores principiantes, y la necesidad de una adecuada disciplina de programación para soslayarlos.

## 7.1 Concepto de subprograma

Un *subprograma*, como su nombre indica, es una parte de un programa. Como mecanismo de programación, un subprograma es una parte de un programa que se desarrolla por separado y se utiliza invocándolo mediante un nombre simbólico.

Desde el punto de vista de una buena metodología de programación, el mecanismo de subprograma debe utilizarse para fragmentos del programa que tengan un cierto sentido en sí mismos. Si se hace así, podríamos decir que, al igual que un programa sirve para resolver un problema, un subprograma sirve para resolver un *subproblema*.

El empleo de subprogramas, desarrollando por separado ciertas partes del programa, resulta especialmente ventajoso en los casos siguientes:

- En programas complejos: Si el programa se escribe todo seguido resulta muy complicado de entender, porque se difumina la visión de su estructura global entre la gran cantidad de operaciones que forman el código del programa (¡los árboles impiden ver el bosque!). Aislando ciertas partes como subprogramas separados se reduce la complejidad de la visión global del programa.
- Cuando se repiten operaciones análogas: Definiendo esa operación como subprograma separado, su código se escribirá sólo una vez, aunque luego se use en muchos puntos del programa. El tamaño total del programa será menor que si se escribiera el código completo de la operación cada vez que se necesita.

La técnica de *refinamientos sucesivos* sugiere descomponer las operaciones complejas de un programa en otras más simples. En sucesivos pasos de refinamiento, cada operación se vuelve a descomponer hasta que todo el programa se puede escribir utilizando las sentencias disponibles en el lenguaje empleado.

Hasta el momento hemos continuado los refinamientos hasta llegar a las sentencias básicas de **C±**. Podemos ver ahora sobre un ejemplo cómo es el programa resultante si las operaciones intermedias se definen como subprogramas en **C±**.

Por ejemplo, consideremos un programa para calcular el perímetro del triángulo formado por tres puntos (A, B y C), según se muestra en la figura 7.1.

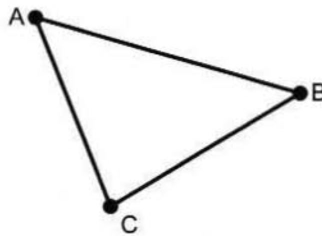



Figura 7.1 Perímetro de un triángulo.

Los primeros pasos del refinamiento serían los siguientes:

*Calcular e imprimir el perímetro* ⟶
    *Leer las coordenadas de los vértices*
    *Calcular el perímetro*
    *Imprimir el perímetro*

A su vez, la operación de lectura de los puntos se puede descomponer en una secuencia de lecturas de las coordenadas de cada uno de los tres puntos:

*Leer las coordenadas de los vértices* ⟶
*Leer coordenadas del punto A*
*Leer coordenadas del punto B*
*Leer coordenadas del punto C*

Definiendo cada subproblema como subprograma, el aspecto del programa, en forma esquemática, sería el siguiente:

```
#include <stdio.h>
  . . . .
void LeerVertices () {
  LeerCoordenadas( A );
  LeerCoordenadas( B );
  LeerCoordenadas( C );
}

void CalcularPerimetro() {
  . . . .
}

void ImprimirPerimetro() {
  . . . .
}

int main() {
  LeerVertices();
  CalcularPerimetro();
  ImprimirPerimetro();
}
```

En este esquema del ejemplo vemos cómo los subprogramas para leer los vértices, calcular el perímetro e imprimir el perímetro aparecen descritos después de la directiva `#include` y antes del inicio del programa principal indicado mediante `int main()`. En esta nueva zona del programa se deben describir todos los subprogramas y el resto de elementos globales que se necesiten en el desarrollo del programa. Así, la estructura para describir un programa en **C±**, más completa que la presentada en el tema 2, sería la siguiente:

*Programa* **::=** **{** *Include* **}** **{** *Declaración_global* **}**
    `int main()` *Bloque*

*Declaración_global* **::=** *Declaración_de_constante* **|**
    *Declaración_de_variable* **|** *Declaración_de_tipo* **|** *Subprograma*

*Subprograma* **::=** *Cabecera_subprograma Bloque*

Como elementos globales de un programa completo en **C±** se pueden declarar constantes y variables en la forma ya estudiada. También se podrán declarar

tipos como se verá en temas posteriores. Además y esto es lo importante en este apartado, un programa completo en **C±** es una colección de subprogramas delimitados por su correspondiente cabecera. Cada subprograma está constituido por su propio *Bloque* de código de manera semejante a un programa completo. En realidad un programa en **C±** es el *programa principal* o "main" cuya cabecera es `int main()`.

A continuación se estudian las dos formas fundamentales de subprogramas en programación imperativa: *funciones* y *procedimientos*, y su manejo utilizando el lenguaje **C±**.

## 7.2 Funciones

Cuando se diseña y desarrolla un programa aparecen con frecuencia operaciones significativas que dan como resultado un valor simple y único en función de ciertos parámetros. Por ejemplo:

| | |
|---|---|
| *Potencia:* | $x^n$ |
| *Volumen de un cubo:* | $lado^3$ |
| *Area de un triángulo:* | $(base * altura)/2$ |
| *Distancia entre dos puntos:* | $((x1 - x2)^2 + (y1 - y2)^2)^{1/2}$ |

Estas operaciones se pueden considerar subprogramas y más exactamente *funciones*. Una función es un tipo de subprograma que calcula como resultado un valor único a partir de otros valores dados como argumentos. En líneas generales, una función se asemeja bastante a la idea matemática de función $F(x, y, ...)$, con argumentos $x$, $y$, ...

### 7.2.1 Definición de funciones

El primer paso en el manejo de una función es declarar su interfaz. Esta declaración incluye su nombre, los argumentos que necesita con el correspondiente tipo para cada uno de ellos, y el tipo de resultado que proporciona. En **C±** esto se realiza escribiendo una *cabecera de función* de la siguiente forma:

*TipoResultado NombreFuncion*`(` *Tipo1 argumento1*`,` *Tipo2 argumento2*`,` ... `)`

La declaración se inicia con el tipo de resultado que proporciona la función y a continuación el nombre de la función, que debe ser un identificador. Seguidamente la lista de los argumentos, entre paréntesis, y separados por el carácter coma (`,`). Las declaraciones de argumentos son similares a las declaraciones

de variables: por cada argumento se declaran su tipo y nombre, separados por al menos un carácter blanco.

Las cabeceras de las funciones para los ejemplos anteriores serían las siguientes:

```
float Potencia( float x, int n )

int VolumenCubo( int lado )

float AreaTriangulo( float base, float altura )

float Distancia( float x1, float y1, float x2, float y2 )
```

Estas cabeceras representan la interfaz entre la definición de la función y su utilización posterior. Los argumentos que aparecen en la cabecera son los *argumentos formales*. No son variables del programa, sino sólo nombres simbólicos que sirven para formalizar la definición posterior de la función, permitiendo hacer referencia a los argumentos en la definición de los cálculos.

La definición completa de una función se compone de una *cabecera* seguida de un *cuerpo de función* que tiene la misma estructura que un *Bloque* de programa completo. Este bloque comienza con una parte declarativa y continúa con una parte ejecutiva. En la parte declarativa se pueden declarar *constantes* y *variables locales* que sólo son visibles en el cuerpo de la función. La parte ejecutiva estará constituida por una secuencia de sentencias.

En las sentencias que constituyen el cuerpo de la función se puede (y se debe) hacer uso de los argumentos formales declarados en su interfaz. Esto permite parametrizar los cálculos de la función para los valores particulares de los argumentos. Así, otra forma de ver las funciones es como *expresiones parametrizadas*.

Por ejemplo, la definición completa de las funciones anteriores se realizaría de la siguiente forma:

```
float Potencia( float x, int n ) {
  float p = 1.0;

  for (int k = 1; k <= n; k++) {
    p = p * x;
  }
  return p;
}

int VolumenCubo( int lado ) {
  return lado*lado*lado;
}
```

```
float AreaTriangulo( float base, float altura ) {
  return (base * altura) / 2.0;
}

float Distancia( float x1, float y1, float x2, float y2 ) {
  float deltaX, deltaY;

  deltaX = x2 - x1;
  deltaY = y2 - y1;
  return sqrtf( deltaX*deltaX + deltaY*deltaY );
}
```

En estos ejemplos se observa la existencia de una nueva sentencia de **C±**, iniciada con la palabra clave `return`. Esta sentencia sirve para devolver como valor de la función el resultado de los cálculos realizados. Esta sentencia tiene la siguiente estructura:

`return` *expresión*;

y provoca la finalización inmediata de la ejecución de la función. El resultado de la expresión debe ser un valor del tipo indicado en la declaración de la función. Dicho valor es el que se devuelve como resultado de la función.

La sentencia `return` se puede insertar en cualquier punto de la parte ejecutable de la función. Además, es posible utilizar más de una sentencia `return` en una misma función. La ejecución de la función acaba cuando se ejecuta cualquiera de las sentencias `return`.

A continuación se muestra la definición de una función con varias sentencias de retorno.

```
int Maximo2(int x, int y) {
  if (x >= y) {
    return x;
  } else {
    return y;
  }
}
```

### 7.2.2 Uso de funciones

Para usar una función en los cálculos de un programa se invoca dicha función escribiendo su nombre y a continuación, entre paréntesis, los valores concretos de los argumentos, separados por comas. Esta invocación de la función representa un valor del tipo de la función, que podrá ser usado como operando en

una expresión aritmética, o en general en cualquier parte del programa en que sea válido escribir una expresión de ese tipo.

Al invocar una función es obligatorio que los valores suministrados para los argumentos (los *argumentos reales*) correspondan en número y tipo con los argumentos en la definición (los *argumentos formales*). La correspondencia de tipo significa que el tipo del argumento en la invocación sea compatible en asignación con el tipo de argumento formal. Por ejemplo:

```
VolumenCubo( ladoCubo ) > 27
valorPotencia = Potencia( base, exponente );
area = AreaTriangulo( Distancia( xA, yA, xB, yB ), medidaAltura ) )
```

El resultado del volumen del cubo es un valor entero y se debe comparar con un valor entero (27). La variable **valorPotencia** tendrá que ser de tipo real, el argumento **base** debe ser de tipo real y el argumento **exponente** debe ser de tipo entero.

El resultado del cálculo de la distancia entre los puntos $A$ y $B$ es de tipo real. En el cálculo del área del triángulo, el argumento para la base debe ser de tipo real. Por tanto, el resultado de la distancia entre $A$ y $B$ se puede utilizar como base del triángulo. La variable **medidaAltura** también debe ser de tipo real.

El efecto de la invocación de una función puede describirse de forma simplificada de la siguiente manera:

1. Se evalúan las expresiones de los valores de los argumentos.
2. Se asignan dichos valores a los correspondientes argumentos formales.
3. Se ejecuta el código de la definición de la función, hasta alcanzar una sentencia de retorno.
4. El valor retornado se usa en el punto donde se invocó la función.

Otros posibles efectos de la invocación de una función o procedimiento se describen más adelante.

### 7.2.3 Funciones predefinidas

Se consideran *funciones predefinidas* las que forman parte del propio lenguaje de programación. Estas funciones están siempre disponibles en cualquier programa. Algunos lenguajes de programación tienen un repertorio más o menos amplio de estas funciones, en particular los que usan funciones con nombre como alternativa a operadores matemáticos para evitar la proliferación de símbolos especiales.

Los lenguajes C y C++ disponen de pocas funciones predefinidas (a cambio ofrecen una gran variedad de operadores). Además estas funciones predefinidas sólo resultan útiles en un uso avanzado del lenguaje. Por ejemplo:

| | |
|---|---|
| `int sizeof(` *tipo* `)` | Tamaño en bytes de un dato del tipo indicado |

Las funciones predefinidas son, en general, *seudofunciones*. Esto es particularmente cierto para las funciones que usan tipos como argumentos o en las que el tipo del argumento no está totalmente determinado (por ejemplo, admiten cualquier tipo numérico).

En **C±** se prescinde del uso de funciones predefinidas, por las consideraciones indicadas anteriormente.

### 7.2.4 Funciones estándar

Al realizar programas en **C±** podremos utilizar funciones que estén definidas en módulos ya redactados de antemano. Algunos módulos constituyen *librerías estándar* y están disponibles en la mayoría de los compiladores de C/C++, tal como ya se dijo al hablar de la operaciones de lectura y escritura o para el manejo de caracteres.

Las funciones definidas en módulos estándar se denominan *funciones estándar* y pueden ser utilizadas sin necesidad de escribir su definición, pero a diferencia de las funciones predefinidas hay que indicar expresamente que se van a utilizar dichos módulos de librería mediante la directiva `#include` del correspondiente módulo que las contenga.

Por ejemplo, tal y como ya se comentó en el tema 2, mediante la directiva `#include <ctype.h>` se pueden utilizar funciones que facilitan el manejo de las diferentes clases de caracteres. Este módulo de librería incluye funciones tales como:

| | |
|---|---|
| `bool isalpha( char c )` | Indica si `c` es una letra |
| `bool isascii( char c )` | Indica si `c` es un carácter ASCII |
| `bool isblank( char c )` | Indica si `c` es un carácter de espacio o tabulación |
| `bool isdigit( char c )` | Indica si `c` es un dígito decimal (0-9) |
| `bool islower( char c )` | Indica si `c` es una letra minúscula |
| `bool isspace( char c )` | Indica si `c` es espacio en blanco o salto de línea o página |
| `bool isupper( char c )` | Indica si `c` es una letra mayúscula |
| `char tolower( char c )` | Devuelve la minúscula correspondiente a `c` |
| `char toupper( char c )` | Devuelve la mayúscula correspondiente a `c` |

El tipo de resultado `bool` sirve para indicar si un resultado es SI/NO y se introducirá formalmente en el próximo tema 9.

En lo referente a funciones matemáticas, se dispone del módulo estándar `math`. Para utilizar sus funciones habrá que incluir la directiva `#include <math.h>` al comienzo del programa. Este módulo dispone de un gran número de funciones matemáticas con nombres distintos dependiendo del tipo del argumento y el tipo de resultado, algunas de ellas son las siguientes:

| | |
|---|---|
| `float sqrtf( float x )` | raíz cuadrada de $x$ |
| `float expf( float x )` | exponencial $e^x$ |
| `float logf( float x )` | logaritmo neperiano de $x$ |
| `float powf( float x, float y )` | potencia $x^y$ |
| `float sinf( float x )` | seno de $x$ |
| `float cosf( float x )` | coseno de $x$ |
| `float tanf( float x )` | tangente de $x$ |
| `float atanf( float x )` | arcotangente de $x$ |
| `float roundf( float x )` | valor de $x$ redondeado a entero |

La función `sqrtf` ya ha sido utilizada en este mismo tema para la definición de la función `Distancia`. Para los objetivos de esta asignatura es suficiente con conocer las funciones aquí relacionadas. En el caso de necesitar alguna función matemática adicional se deberá consultar el correspondiente manual de C/C++.

Queda fuera del ámbito de este libro la enumeración de todos los módulos y funciones de librería existentes en C/C++. Algunas de ellas se irán presentado en el momento en que se introduzcan los correspondientes conceptos. Para cualquier consulta adicional se debería hacer uso de un manual de C/C++ .

## 7.3 Procedimientos

Un *procedimiento* es un subprograma que realiza una determinada acción. A diferencia de las funciones, un procedimiento no tiene como objetivo, en general, devolver un valor obtenido por cálculo.

Un procedimiento es una forma de subprograma que agrupa una sentencia o grupo de sentencias que realizan una acción, y permite darles un nombre por el que se las pueden identificar posteriormente. Estas sentencias, al igual que en las funciones, se pueden parametrizar mediante una serie de argumentos. Así, otra forma de ver a los procedimientos es como *acciones parametrizadas*.

Por ejemplo, durante el desarrollo de un programa podemos identificar acciones tales como:

*Trazar una línea de longitud dada*
*Imprimir el resultado*
*Ordenar dos valores*
*Leer las coordenadas de un punto*

Si nos interesa, podremos definir estas acciones como procedimientos, y luego invocarlas en el programa cuando se necesite.

### 7.3.1 Definición de procedimientos

La definición en **C±** de un procedimiento es prácticamente igual a la de una función:

`void` *NombreProcedimiento*`(` *Tipo1 argum1*`,` *Tipo2 argum2*`,` ... `)` *Bloque*

La diferencia principal es que no se declara el tipo de valor del resultado, ya que no existe dicho valor. La palabra reservada **void** es la que indica que no hay resultado de ningún tipo. Además, con cierta frecuencia interesa definir procedimientos sin argumentos. En estos casos sólo es necesario dar el nombre, y no habrá lista de argumentos entre los paréntesis.

Como ejemplo, podemos dar posibles definiciones de procedimientos que correspondan a las dos primeras acciones citadas anteriormente:

```
void TrazarLinea( int longitud ) {
  for (int k = 1; k <= longitud; k++) {
    printf( "-" );
  }
}

void EscribirResultado() {
  printf( "Resultado:%10f\n", resultado );
}
```

En el primer caso se trata de un procedimiento para trazar una línea horizontal de cualquier longitud, a base de imprimir guiones. El resultado de este procedimiento no es un valor determinado, sino la acción de trazado de la línea. En el segundo caso el procedimiento se utiliza solamente para imprimir con un formato concreto un resultado ya calculado.

Si se desea, en la definición de un procedimiento pueden usarse también sentencias de retorno, pero con un significado algo diferente que en el caso de las funciones. La sentencia

```
return;
```

se escribe ahora simplemente así, sin ninguna expresión que la acompañe, ya que no hay ningún valor que devolver. Esta sentencia sirve simplemente para terminar la ejecución del procedimiento en ese momento, y volver al punto siguiente a donde se invocó. Por ejemplo, otra posible definición del procedimiento de imprimir un resultado sería:

```
void EscribirResultado() {
  if (resultado < 0) {
    printf( "Problema no resuelto" );
    return;
  }
  printf( "Resultado:%10f\n", resultado );
}
```

En este caso si la condición de la sentencia `if` resulta cierta la sentencia final de escritura no se ejecutará, ya que la sentencia de retorno termina la acción del procedimiento de forma inmediata.

### 7.3.2 Uso de procedimientos

Para usar un procedimiento hay que invocarlo. Dicha invocación o llamada constituye por sí sola una sentencia de **C±**, cuyo formato es:

*NombreProcedimiento*`(` *argumento1*`,` *argumento2*`, ... );`

Como puede observarse, un procedimiento se invoca escribiendo su nombre y a continuación, si los hay, los valores de los argumentos particulares en esa llamada, separados por comas. Los valores de los argumentos pueden darse, en general, mediante expresiones. Si no hay argumentos no se suprimen los paréntesis.

Los argumentos en la llamada (*argumentos reales*) deberán ser compatibles con los indicados en la declaración (*argumentos formales*), tal como se dijo para las funciones. Por ejemplo, los procedimientos declarados anteriormente podrían invocarse de la forma:

```
TrazarLinea( 3*Lado );
EscribirResultado();
```

Con la primera llamada se trazará una línea con el triple de la longitud del `Lado`. Con la segunda llamada se escribirá el resultado según el formato programado en la definición de este procedimiento.

De forma simplificada, la invocación de un procedimiento produce un efecto análogo a la secuencia de acciones siguientes:

1. Se evalúan las expresiones de los valores de los argumentos.
2. Se asignan dichos valores a los correspondientes argumentos formales.
3. Se ejecuta el código de la definición del procedimiento, hasta alcanzar el final del bloque o una sentencia de retorno.
4. El programa que invocó al procedimiento continúa en el punto siguiente a la sentencia de llamada.

### 7.3.3 Procedimientos estándar

Al igual que para las funciones, en los *módulos estándar* asociados a cada compilador de C/C++ se dispone de diversos *procedimientos estándar* que pueden utilizarse sin más que hacer uso de la directiva `#include` del correspondiente módulo.

En particular ya se han mencionado y utilizado los procedimientos estándar de lectura de datos o escritura de resultados, que están disponibles tras incluir la cabecera <stdio.h>. Otros procedimientos estándar se irán introduciendo en los siguientes temas, o pueden consultarse en un manual de C/C++.

## 7.4 Paso de argumentos

La manera fundamental de comunicar información entre las sentencias de un subprograma y el programa que lo utiliza es mediante los argumentos. En **C±** existen dos formas distintas de realizar esta comunicación, que se denominan *paso de argumentos por valor* y *paso de argumentos por referencia*, que se describen a continuación.

### 7.4.1 Paso de argumentos por valor

Esta es la forma utilizada en los ejemplos que se han mencionado hasta el momento. Los argumentos representan valores que se transmiten desde el programa que llama hacia el subprograma. En el caso de las funciones hay además un valor de retorno, que es el valor de la función que se transmite desde el subprograma hacia el programa que lo llamó.

Los argumentos reales en la llamada al subprograma pueden darse en general en forma de expresiones, cuyos tipos de valor deben ser compatible en asignación con los tipos de los argumentos formales.

El modo de paso por valor implica que los elementos usados como argumentos en la llamada al subprograma no pueden ser modificados por la ejecución de

las sentencias del subprograma. Esto es cierto incluso en el caso de que en el subprograma se ejecuten asignaciones a los argumentos formales, considerados como variables locales dentro del subprograma.

El *paso de argumentos por valor* puede describirse de la siguiente manera:

1. Se evalúan las expresiones de los argumentos reales usados en la llamada.
2. Los valores obtenidos se copian en los argumentos formales.
3. Los argumentos formales se usan como variables dentro del subprograma. Si a estas variables se les asignan nuevos valores, no se estará modificando el argumento real, sino sólo la copia.

Por ejemplo, se puede modificar la definición de la función para el cálculo de la distancia entre dos puntos de la siguiente forma:

```
float Distancia( float x1, float y1, float x2, float y2 ) {
  x1 = x2 - x1;
  y1 = y2 - y1;
  return sqrt( x1*x1 + y1*y1 );
}
```

Como puede verse, dentro del procedimiento se asignan nuevos valores a algunos de los argumentos. Pese a ello, un fragmento de programa tal como:

```
xA = 23.5; yA = 12.3;
xB = 5.7; yB = 2.6;
distanciaAB = Distancia( xA, yA, xB, yB );
```

no modifica las variables externas `xA` e `yA` usadas como argumentos, que mantienen los valores que tenían antes de la llamada.

Puesto que la reasignación de valor a un argumento pasado por valor resulta algo confusa, es preferible evitar esta circunstancia todo lo posible. Esta es una regla recomendable en el *Manual de Estilo* de esta asignatura.

### 7.4.2 Paso de argumentos por referencia

En ciertos casos es deseable que el subprograma pueda modificar las variables que se usen como argumentos. Esto permite producir simultáneamente varios resultados y no sólo uno. El mecanismo de paso por valor no permite que el subprograma modifique directamente una variable pasada como argumento. Para conseguirlo hay que usar el otro modo de paso de argumentos, denominado *paso de argumentos por referencia*.

El paso de un argumento por referencia se indica en la cabecera del subprograma, anteponiendo el símbolo `&` al nombre del argumento formal, de la siguiente manera:

*TipoResultado Nombre*( *TipoArgumento* & *argumento*, ... )

Si un argumento se pasa por referencia ya no será válido usar como argumento real una expresión. El argumento real usado en la llamada debe ser necesariamente una variable del mismo tipo. Esta variable será utilizada en el subprograma como si fuera suya, es decir, la asignación de nuevo valor al argumento modifica realmente la variable externa pasada como argumento.

El paso de argumentos por referencia puede describirse de la siguiente manera:

1. Se seleccionan las variables usadas como argumentos reales.
2. Se asocia cada variable con el argumento formal correspondiente.
3. Se ejecutan las sentencias del subprograma como si los argumentos formales fuesen los argumentos reales.

Ahora se pueden escribir las definiciones como subprograma de las restantes acciones puestas como ejemplo al hablar de procedimientos en el apartado 7.3.

*Ordenar dos valores*
*Leer las coordenadas de un punto*

En ellas necesitamos utilizar argumentos pasados por referencia en los que se puedan dejar los valores ordenados o las coordenadas leídas.

```
void OrdenarDos( int & y, int & z ) {
  int aux;

  if (y > z) {
    aux = y;
    y = z;
    z = aux;
  }
}

void LeerCoordenadas( char Punto, float & x, float & y ) {
  printf( "Punto %c\n",Punto );
  printf( "¿Coordenada X ?" );
  scanf( "%f",&x);
  printf( "¿Coordenada Y ?" );
  scanf( "%f",&y );
  printf( "\n" );
}
```

Un ejemplo de uso sería:

```
OrdenarDos( A, B );
LeerCoordenadas( 'A', xA, yA );
```

La notación & usada en las cabeceras de subprogramas para indicar el paso por referencia es una novedad de C++ respecto al lenguaje C, y evita tener que recurrir al uso explícito de punteros para permitir el paso por referencia, simplificando así las invocaciones de los suprogramas.

Excepcionalmente, en los ejemplos de este libro se usan algunos subprogramas estándar de C, en particular el procedimiento predefinido de lectura `scanf` del módulo `stdio`, que sí requieren el paso de punteros como argumentos. Obsérvese que este caso hay que usar explícitamente el operador & (que obtiene la dirección de una variable) cada vez que se invoca ese subprograma. Los punteros se introducirán en el tema 13 de este libro.

## 7.5 Visibilidad. Estructura de bloques

Como norma general, en un punto determinado de un programa se pueden utilizar todos los elementos definidos con anterioridad. Podemos decir que desde un punto del programa se pueden "ver" los elementos definidos hasta ese momento. Esta regla sencilla de visibilidad se amplía con nuevas limitaciones cuando se definen subprogramas. La definición de un subprograma está formada por una *cabecera* o interfaz, y un *bloque de código* que es el cuerpo del subprograma. Ese bloque de código constituye una barrera de *visibilidad* que hace que los elementos declarados en el interior del cuerpo de un subprograma no sean visibles desde el exterior.

Es decir, la definición de un subprograma construye un nuevo elemento, utilizable en el resto del código, y al mismo tiempo realiza una *ocultación* de sus detalles de realización. Un programa que contenga subprogramas tiene, por tanto, una estructura de bloques que marcan ámbitos de visibilidad, tal como se muestra en el ejemplo de la figura 7.2, donde se han marcado los diferentes bloques y algunos puntos concretos dentro de ellos.

Los elementos definidos en el ámbito más externo son elementos *globales*, mientras que los elementos definidos en el interior del bloque de un subprograma son elementos *locales* a dicho subprograma. Cada bloque es completamente opaco desde el exterior y se puede considerar como una caja negra. Los elementos locales de un bloque son invisibles desde el exterior del bloque y dejan de existir al finalizar la ejecución del bloque.

La cabecera de un subprograma se encuentra en la frontera entre el interior y el exterior de dicho subprograma, y es parcialmente visible desde el exterior. Desde el interior del subprograma la cabcera es visible en todos sus detalles. Desde el exterior es visible el nombre y tipo del subprograma, y el tipo de cada

```
#include <stdio.h>                                -- global ---------.
const float Pi = 3.1416;                                             |
                                         <A>                         |
void Cambiar( int & v1, int & v2 ) {              -- Cambiar ----.   |
  int temp;                                                      |   |
                                         <B>                     |   |
  temp = v1;                                                     |   |
  v1 = v2;                                                       |   |
  v2 = temp;                                                     |   |
}                                                 ---------------'   |
                                                                     |
int Factorial( int numero ) {                     -- Factorial --.   |
  int fac = 1;                                                   |   |
                                                                 |   |
  for (int k=2; k<=numero; k++) {                 -- for ----.   |   |
    fac = fac * k;                       <C>                 |   |   |
  }                                               -----------'   |   |
  return fac;                            <D>                     |   |
}                                                 ---------------'   |
                                         <E>                         |
int main() {                                      -- main -------.   |
  float r;                                                       |   |
                                         <F>                     |   |
  scanf( "%f", r );                                              |   |
  printf( "Area = %f, AreaC( r ) );                              |   |
}                                                 ---------------'---'
```

Figura 7.2 Ejemplo de bloques en un programa en **C±**.

argumento, pero no su nombre particular. La vista externa de la cabecera es realmente la interfaz del subprograma. El contenido lógico de la interfaz es lo que se denomina *signatura* del subprograma, que es suficiente para comprobar si la invocaciones son consistentes con su definición:

| Cabecera (código real) | Signatura (vista lógica) |
|---|---|
| `void Cambiar( int & v1, int & v2 )` | `void Cambiar( int &, int & )` |
| `int Factorial( int numero )` | `int Factorial( int )` |

Finalmente hay que mencionar el caso especial de la sentencia `for` de **C±**. En ella se declara la variable contador del bucle en la misma sentencia, y establece un ámbito local limitado en el cual es visible dicha variable contador. De hecho al terminar la ejecución del bucle la variable contador desaparece.

Como resumen, y para ilustrar con un ejemplo concreto las reglas de visibilidad, en la tabla siguiente se recogen las listas de elementos que son visibles en puntos concretos del programa de ejemplo de la Figura 7.2:

| **Punto** | **Elementos visibles** |
|---|---|
| A | Pi |
| B | Pi, Cambiar, v1, v2, temp |
| C | Pi, Cambiar, Factorial, numero, fac, k |
| D | Pi, Cambiar, Factorial, numero, fac |
| E | Pi, Cambiar, Factorial |
| F | Pi, Cambiar, Factorial, main, r |

## 7.6 Recursividad de subprogramas

Un caso especial que no ha sido considerado hasta este momento es la posibilidad de que en un subprograma se haga uso de ese mismo subprograma. Cuando un subprograma hace una llamada a sí mismo se dice que es un subprograma recursivo. Un estudio más detallado y completo del concepto de *recursividad* queda fuera de los objetivos de este libro y sería objeto de estudio en un curso más avanzado.

Para ilustrar de manera sencilla la recursividad utilizaremos el algoritmo para el cálculo del factorial de un número natural. En el tema 5 se decía que el factorial de un número $n$ se puede calcular mediante la fórmula:

$$n! = 1 \times 2 \times 3 \times 4 \times ... \times n$$

Este cálculo se puede también expresar de manera recursiva mediante el siguiente algoritmo (para el caso general):

$$n! = n \times (n-1)!$$

La ejecución de un algoritmo recursivo debe terminar en algún momento. Para ello hace falta que las llamadas recursivas, internas, del subprograma a sí mismo deben estar controladas en una estructura condicional. Se distinguen diferentes casos en la invocación del subprograma dependiendo de los valores concretos de los argumentos. En algunos casos, denominados *casos minimales*, la solución se obtiene directamente, sin necesidad de llamadas recursivas. En el resto de los casos, que denominaremos *casos generales*, se necesitan invocaciones recursivas para obtener la solución.

En el cálculo del factorial los casos minimales corresponden a:

$$0! = 1 \quad \text{y} \quad 1! = 1$$

A continuación se muestra el listado de un programa recursivo para el cálculo del factorial utilizando la función recursiva `FactorialRecursivo`.

```
/**********************************************************
* Programa: FactorialRecursivo
*
* Descripción:
*   Este programa calcula el factorial de los primeros
*   números naturales, de forma recursiva
**********************************************************/
#include <stdio.h>

int FactorialRecursivo( int n ) {
  if (n <= 1) {
    return 1;
  } else {
    return n * FactorialRecursivo( n-1 );
  }
}

int main() {
  for (int i = 0; i <= 10; i++) {
    printf( "%2d! vale:%10d\n", i, FactorialRecursivo( i ) );
  }
}
```

La ejecución del programa produce el siguiente resultado:

```
 0! vale:         1
 1! vale:         1
 2! vale:         2
 3! vale:         6
 4! vale:        24
 5! vale:       120
 6! vale:       720
 7! vale:      5040
 8! vale:     40320
 9! vale:    362880
10! vale:   3628800
```

## 7.7 Problemas en el uso de subprogramas

El empleo de funciones y procedimientos, las dos formas de paso de argumentos, las reglas de visibilidad entre bloques, etc., ofrecen interesantes posibi-

lidades para el desarrollo de los programas. Sin embargo, un uso incorrecto de estas posibilidades puede dar lugar a ciertos problemas. En este apartado se analizan algunos de estos problemas y se dan las directrices para poderlos evitar.

### 7.7.1 Uso de variables globales. Efectos secundarios

Ya se ha comentado que uno de los objetivos de la programación es la claridad del código desarrollado, es decir, que sea fácil de entender. En el caso de los subprogramas una cualidad deseable para ello es lo que se denomina *transparencia referencial*, que consiste en que el efecto de una llamada al subprograma pueda predecirse simplemente con la información contenida en el código de la llamada. Dicho de otro modo, siempre que se invoque al subprograma con los mismos valores de los argumentos se debe obtener el mismo resultado.

La transparencia referencial se garantiza si el código del subprograma utiliza solamente elementos mencionados en la lista de argumentos o definidos como elementos locales. La posibilidad de utilizar directamente por su nombre variables globales, que no se mencionan en la lista de argumentos, permite escribir subprograma que carecen de transparencia referencial. Cuando un subprograma modifica alguna variable externa, se dice que está produciendo *efectos secundarios* o laterales (en inglés *side effects*). El uso de subprogramas con efectos secundarios debe hacerse con precaución.

La transparencia referencial es deseable tanto para las funciones como para los procedimientos. Sin embargo, para las funciones es una cualidad casi imprescindible. Resulta muy difícil de justificar y comprender que se produzcan efectos secundarios al utilizar una función dentro de una expresión. Idealmente, además todos los argumentos de las funciones deberían pasarse por valor, dado que el único resultado deseable de la función es devolver el valor de la propia función. Una función que no produzca efectos laterales y todos sus argumentos se pasen por valor se dice que es una *función pura*.

El acceso directo a variables globales puede suplantar el uso de argumentos, de manera que en un caso extremo podemos escribir subprogramas sin ningún argumento:

```
float base, altura;
---
float AreaRectangulo () {
  return base * altura;
}
---
```

```
base = 3.0;
altura = 2.5
printf( "%f", AreaRectangulo() );
```

En general es preferible mencionar expresamente en la lista de argumentos todos los elementos externos que intervienen en la operación de un subprograma. El ejemplo anterior resulta mucho más natural si escribirmos:

```
float AreaRectangulo( float base, float altura ) {
  return base * altura;
}
...
printf( "%f", AreaRectangulo( 3.0, 2.5 ) );
```

Aunque la transparencia referencial es claramente deseable, hay algunos casos en que el uso directo de variables globales puede tener alguna ventaja. Esto ocurre, en general, si algunos de los argumentos son siempre los mismos en prácticamente todas las llamadas al subprograma. Si esos argumentos se almacenan en variables globales y se eliminan de la lista de argumentos, el código de las llamadas se simplifica. Como ejemplo plantearemos ir acumulando una serie de valores. El código normal sería:

```
void Acumular( float & suma, float valor ) {
  suma = suma + valor;
}
...
float total;
...
total = 0;
Acumular( total, 3.5 );
Acumular( total, 0.2 );
Acumular( total, 2.7 );
Acumular( total, 1.5 );
printf( "%d", total );
```

Usando una variable global las llamadas resultan algo más sencillas:

```
float total;
...
void Acumular( float valor ) {
  total = total + valor;
}
...
total = 0;
Acumular( 3.5 );
Acumular( 0.2 );
```

```
Acumular( 2.7 );
Acumular( 1.5 );
printf( "%d", total );
```

Conviene insistir en que el empleo de subprogramas con efectos laterales debe hacerse siempre con precaución, y limitarlo a casos especiales como el de este ejemplo, en el que el primer argumento del código normal, sin efectos laterales, sería siempre exactamente el mismo. Las siguientes secciones muestran con claridad algunos peligros del uso directo de variables globales.

## 7.7.2 Redefinición de elementos

Dentro de cada bloque se pueden definir elementos locales dándoles el nombre que se considere más adecuado en cada caso. Los nombres locales no afectan al código fuera del bloque, ya que no son visibles. Incluso es posible repetir el mismo nombre para elementos diferentes definidos en distintos bloques. Por ejemplo, es habitual utilizar las variables de nombres `i`, `j`, `k`, etc. para los índices de las sentencias de iteración. Si estos nombres repetidos están en distintos bloques no existe ningún problema pues en cada uno de ellos estas variables tienen un carácter local que no afecta a las definiciones de los otros bloques.

Sin embargo cuando en el interior de un bloque se define un elemento local con el mismo nombre que otro elemento global la situación es algo distinta. De acuerdo con las reglas de visibilidad, cualquier bloque tiene acceso a todos los elementos globales. Sin embargo, al dar un nombre ya utilizado como global a un nuevo elemento local del bloque se está *redefiniendo* dicho nombre, y automáticamente se pierde la posibilidad de acceso al elemento global del mismo nombre. Se dice que el nombre local oculta o "hace sombra" (en inglés *shadow*) al nombre global. Por ejemplo, en el siguiente programa:

```
/** Programa: Shadow */
/* Ejemplo de ocultación de nombres globales */
#include <stdio.h>

int Global;
int Exterior = 50;

void Interior() {
  const int Exterior = 30;

  Global = Exterior*Exterior;
}
```

```
void Secundario() {
  Interior();
  Exterior = Global/2;
}

void Primario( int & Exterior, int & Interior ) {
  Secundario();
  Interior = Exterior - 5;
}

int main() {
  int Interior = 40;

  Primario( Interior, Interior );
  printf( "%5d%5d%5d\n", Global, Exterior, Interior );
}
```

el cálculo de **Global** en el procedimiento **Interior** utiliza su constante local **Exterior** con valor igual a 30. El procedimiento **Secundario** utiliza el procedimiento **Interior** y la variable global **Exterior**. Solamente dentro del procedimiento **Primario** se tiene acceso a los argumentos **Interior** y **Exterior** del mismo, que están ocultando la variable global **Exterior** y el procedimiento **Interior**. La variable **Interior** del programa principal inicializada a 40 oculta el procedimiento **Interior** y además se modifica en la llamada con doble referencia al procedimiento **Primario**.

Aunque el programa es bastante extraño es completamente correcto. Como ejercicio, resulta interesante hacer un análisis paso a paso de su ejecución. Para la correspondiente comprobación, la ejecución del programa da el siguiente resultado:

```
900  450   35
```

Es evidente que el empleo de elementos diferentes con el mismo nombre aumenta la complejidad del programa y se dificulta mucho su comprensión. Por otro lado, se abre una vía de errores no siempre detectables en compilación. Aunque se pretenda utilizar un símbolo como local, si se olvida que es un nombre redefinido se asumirá incorrectamente el significado dado como símbolo externo. Por tanto, salvo que sea imprescindible, no se debe utilizar la *redefinición de elementos*.

### 7.7.3 Doble referencia

Se produce *doble referencia* (en inglés *aliasing*) cuando una misma variable se referencia con dos nombres distintos, cosa que puede ocurrir en la invocación de subprogramas con argumentos pasados por referencia. Fundamentalmente, esto puede ocurrir en dos situaciones muy concretas:

1. Cuando un subprograma utiliza una variable externa que también se le pasa como argumento.
2. Cuando para utilizar un subprograma se pasa la misma variable en dos o más argumentos.

Habitualmente, un subprograma se escribe pensando que todos sus argumentos son distintos y que nunca coincidirán con ninguna variable externa ya utilizada dentro de él. La doble referencia produce resultados incomprensibles a primera vista. Consideremos, por ejemplo, el siguiente fragmento de programa:

```
int global;

void Cuadrado( int & dato ) {
  global = 5;
  dato = dato * dato;
}

global = 3;
Cuadrado( global );
```

Después de la ejecución del procedimiento `Cuadrado( global )` la variable `global` tiene un valor igual a 25. Esto es debido a que el procedimiento `Cuadrado` utiliza directamente como `dato` la variable `global` pasada por referencia. En el momento del cálculo del producto, el valor de dicha variable es 5 y por tanto el producto por sí misma es 25.

Una situación similar se puede producir con el siguiente procedimiento:

```
void CuadradoCubo( int & x, int & x2, int & x3 ) {
  x2 = x*x;
  x3 = x*x*x;
}
```

Si se utiliza con los siguientes argumentos y valores:

```
A = 4;
CuadradoCubo( A, A, B );
```

después de la ejecución de este fragmento, los valores de las variables son `A` igual a 16 y `B` igual a 4096.

Como conclusión se puede decir que no se debe utilizar la doble referencia, salvo que el subprograma se diseñe pensando en esa posibilidad. Esto último deberá quedar claro en los comentarios del subprograma.

## 7.8 Ejemplos de programas

Para finalizar este tema se muestran tres programas completos y sus correspondientes resultados. Estos programas utilizan algunas de las funciones y procedimientos que han sido desarrollados a lo largo de todo este tema.

### 7.8.1 Ejemplo: Raíces de una ecuación de segundo grado

Con este programa se trata de calcular las raíces de una ecuación de segundo grado:

$$ax^2 + bx + c = 0$$

las raíces pueden ser reales o imaginarias. Los coeficientes $a$, $b$ y $c$ serán reales y se leerán del teclado. El programa tendrá en cuenta los siguientes casos:

- Si $a$, $b$ y $c$ son iguales a cero: Se considerará una ecuación no válida.
- Si $a$ y $b$ son iguales a cero: La solución es imposible.
- Si $a$ es igual a cero: Una única raíz.
- Si $a$, $b$ y $c$ son distintos de cero: Dos raíces reales o imaginarias.

En este último caso, el cálculo de las raíces se realiza mediante la fórmula:

$$raices = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Para la lectura de los coeficientes conviene utilizar un procedimiento, y así se pueden leer los 3 coeficientes de igual manera. En el siguiente programa está recogido el listado completo.

```
/*********************************************************
* Programa: Raices
*
* Descripción:
*   Este programa calcula las raíces de una
*   ecuación de segundo grado: ax² + bx + c
*********************************************************/
#include <stdio.h>
#include <math.h>
```

```
/** Función para calcular el discriminante */
float Discriminante( float a, float b, float c ) {
  return b*b - 4.0*a*c;
}

/** Procedimiento de lectura de un coeficiente */
void LeerValor( int grado, float & valor ) {
  printf( "¿Coeficiente de grado %1d? ", grado );
  scanf( "%f",&valor );
}

/** Programa principal */
int main() {
  float valorA, valorB, valorC;  /* Coeficientes de la ecuación */
  float parteUno, parteDos;      /* Variables intermedias de cálculo */
  float valorD;                  /* Discriminante de la ecuación */

  LeerValor( 2, valorA );
  LeerValor( 1, valorB );
  LeerValor( 0, valorC );
  if (valorA == 0.0) {
    if (valorB == 0.0) {
      if (valorC == 0.0) {
        printf( "Ecuación no válida\n" );
      } else {
        printf( "Solución imposible\n" );
      }
    } else {
      printf( "Raíz única = %10.2f\n", -valorC/valorB );
    }
  } else {
    parteUno = - valorB/(2.0*valorA);
    valorD = Discriminante( valorA, valorB, valorC );
    if (valorD >= 0.0) {
      parteDos = sqrt(valorD)/(2.0*valorA);
      printf( "Raíces reales :\n" );
      printf( "%10.2f y \n", parteUno+parteDos );
      printf( "%10.2f \n", parteUno-parteDos );
    } else {
      parteDos = sqrt(-valorD)/(2.0*valorA);
      printf( "Raíces complejas :\n" );
      printf( "Parte real =       %10.2f y\n", parteUno );
      printf( "Parte imaginaria = %10.2f \n", parteDos );
    }
  }
}
```

El resultado obtenido por el programa para la ecuación $x^2 + 2x + 2 = 0$ es el siguiente:

```
¿Coeficiente de grado 2? 1.0
¿Coeficiente de grado 1? 2.0
¿Coeficiente de grado 0? 2.0
Raíces complejas :
Parte real =            -1.00 y
Parte imaginaria =       1.00
```

### 7.8.2 Ejemplo: Ordenar tres valores

Este programa es una versión mejorada del mostrado en el tema 5. En este caso se utiliza el procedimiento para la ordenación de dos datos que fue desarrollado en el apartado 7.4 de este mismo tema. Además, se utiliza un procedimiento para leer uno a uno los tres datos a ordenar. El programa permanece en un bucle hasta que se indica que no se necesita ordenar más datos.

A continuación se recoge el listado completo.

```
/************************************************
* Programa: Ordenar3b
*
* Descripción:
*   Este programa ordena 3 valores
************************************************/
#include <stdio.h>
#include <math.h>

/** Procedimiento para ordenar dos datos */
void OrdenarDos( float & y, float & z ) {
  float aux;

  if (y > z) {
    aux = y;
    y = z;
    z = aux;
  }
}

/** Procedimiento para leer un dato */
void LeerDato( int indice, float & dato ) {
  printf( "¿Dato %1d? ", indice );
  scanf( "%f",&dato );
```

```
}

/** Programa principal */
int main() {
  float valorUno, valorDos, valorTres;     /* valores a ordenar */
  char tecla;                              /* tecla de opcion */

  tecla = 'S';
  while (tecla != 'N') {
    /*-- Leer los datos --*/ {
      LeerDato( 1, valorUno );
      LeerDato( 2, valorDos );
      LeerDato( 3, valorTres );
    }
    /*-- Ordenar los datos --*/ {
      OrdenarDos( valorUno, valorDos );
      OrdenarDos( valorUno, valorTres );
      OrdenarDos( valorDos, valorTres );
    }
    /*-- Escribir resultados --*/ {
      printf( "\nDatos Ordenados = \n" );
      printf( "%7.2f %7.2f %7.2f\n", valorUno, valorDos, valorTres );
    }
    /*-- Comprobar si se continúa --*/ {
      tecla = ' ';
      printf( "\n¿Desea continuar(S/N)? " );
      while ((tecla != 'S') && (tecla != 'N')) {
        scanf( "%c", &tecla );
      }
    }
  }
}
```

Los resultados obtenidos en dos ordenaciones consecutivas son los siguientes:

```
¿Dato 1? 12
¿Dato 2? 3
¿Dato 3? 89

Datos Ordenados =
   3.00   12.00   89.00

¿Desea continuar(S/N)? S
¿Dato 1? 9
¿Dato 2? 34
¿Dato 3? 2
```

```
Datos Ordenados =
   2.00    9.00   34.00

¿Desea continuar(S/N)? N
```

### 7.8.3 Ejemplo: Perímetro de un triángulo

Este programa ha sido desarrollado casi completamente a lo largo de este tema. En este apartado se trata solamente de mostrar su estructura global, en la que se aprecia el uso de variables globales entre los distintos procedimientos y funciones y el programa principal. A continuación se recoge el listado completo.

```
/****************************************************
* Programa: Perimetro
*
* Descripción:
*   Programa para calcular el perímetro de un
*   triángulo dado por sus tres vértices
****************************************************/
#include <stdio.h>
#include <math.h>

/*==================*\
  Variables globales
\*==================*/
float xA, yA, xB, yB, xC, yC;   /* Coordenadas de los puntos */
float perimetro;                /* Valor del perimetro */

/*=========================================================*\
  Procedimiento para leer las coordenadas X e Y de un punto.
  Para facilitar la identificación del punto, se tiene que
  pasar la letra que lo identifica como argumento
\*=========================================================*/
void LeerCoordenadas( char Punto, float & x, float & y ) {
  printf( "Punto %c\n",Punto );
  printf( "¿Coordenada X ? " );
  scanf( "%f", &x );
  printf( "¿Coordenada Y ? " );
  scanf( "%f", &y );
  printf( "\n" );
}
```

```
/*=========================================================*\
  Procedimiento para leer las coordenadas de los 3 vértices
\*=========================================================*/
void LeerVertices() {
  LeerCoordenadas( 'A', xA, yA );
  LeerCoordenadas( 'B', xB, yB );
  LeerCoordenadas( 'C', xC, yC );
}

/*======================================*\
  Función para calcular la distancia que hay
  entre dos puntos (x1,y1) y (x2,y2)
\*======================================*/
float Distancia( float x1, float y1, float x2, float y2 ) {
  float deltaX, deltaY;

  deltaX = x2 - x1;
  deltaY = y2 - y1;
  return sqrt( deltaX*deltaX + deltaY*deltaY );
}

/*============================================================*\
  Procedimiento para calcular el perímetro de un triágulo
  NOTA: Se utilizan variables globales dado el excesivo
        número de argumentos necesarios: Total 7 argumentos:
          3 puntos x 2 coordenada = 6 argumentos por valor
          Resultado en perimetro  = 1 argumento por referencia
\*============================================================*/
void CalcularPerimetro() {
  float ladoAB, ladoAC, ladoBC;

  ladoAB = Distancia( xA, yA, xB, yB );
  ladoAC = Distancia( xA, yA, xC, yC );
  ladoBC = Distancia( xB, yB, xC, yC );
  perimetro = ladoAB + ladoAC + ladoBC;
}

/*=========================================================*\
  Procedimiento para imprimir la variable global perímetro
\*=========================================================*/
void ImprimirPerimetro() {
  printf( "El Perímetro es igual a %5.2f\n", perimetro );
}
```

```
/*==================*\
  Programa principal
\*==================*/
int main() {
  LeerVertices();
  CalcularPerimetro();
  ImprimirPerimetro();
}
```

Un ejemplo del resultado de su ejecución es el siguiente:

```
Punto A
¿Coordenada X ? 3.0
¿Coordenada Y ? 0.0

Punto B
¿Coordenada X ? 0.0
¿Coordenada Y ? 0.0

Punto C
¿Coordenada X ? 0.0
¿Coordenada Y ? 4.0

El Perímetro es igual a 12.00
```

# Tema 8

# Metodología de Desarrollo de Programas (III)

Este tema completa el anterior, ampliando la metodología de desarrollo por refinamientos sucesivos con la posibilidad de usar subprogramas como técnica de *abstracción*.

A nivel metodológico, las funciones y procedimientos introducen la posibilidad de descomposición de un problema en subproblemas realmente independientes. Por el momento se mantiene la visión del programa como una sola unidad de compilación (un solo módulo). Más adelante, en el tema 15, se explicará la manera de descomponer un programa en varios módulos compilables por separado.

## 8.1 Operaciones abstractas

Los subprogramas constituyen un primer paso hacia la metodología de programación basada en abstracciones. Los subprogramas permiten definir operaciones abstractas. El siguiente paso será la definición de tipos abstractos de datos, que se introducirán brevemente más adelante, en el tema 14.

Una abstracción es una visión simplificada de una cierta entidad, de la que sólo consideramos sus elementos esenciales, prescidiendo en lo posible de los detalles. Las entidades que podemos abstraer para materializarlas como subprogramas son, en general, operaciones. Con la palabra *operación* englobamos tanto la idea de *acción* como la de *función.*

### 8.1.1 Especificación y realización

Al plantear operaciones abstractas habremos de definir dos posibles visiones. La visión abstracta o simplificada, y la visión detallada, completa. La visión abstracta es la que permite usar dicha operación sin más que conocer *qué hace* dicha operación. La visión detallada es la que define *cómo se hace* dicha operación, y permite que el procesador la ejecute. La primera visión representa el punto de vista de quienes han de utilizar la operación. Se dice que esa visión abstracta es la especificación o *interfaz* de la operación. La visión detallada representa el punto de vista de quien ha de ejecutar dicha acción, y se dice que expresa su *realización* o *implementación*. Resumiendo:

*Especificación*: Qué hace la operación (punto de vista de quien la invoca).

*Realización*: Cómo se hace la operación (punto de vista de quien la ejecuta).

En su forma más sencilla la especificación o interfaz consiste simplemente en indicar cuál es el nombre de la operación y cuáles son sus argumentos. En **C±** la especificación puede ser simplemente una *cabecera de subprograma*.

Esa forma simplificada de especificación indica solamente cuál ha de ser la *sintaxis* o forma de uso de la operación. La especificación completa debe establecer también cuál es la *semántica* o significado de la operación. Para ello podemos añadir un comentario en que se indique qué relación hay entre los argumentos y el resultado de la operación.

La realización, por su parte, debe suministrar toda la información necesaria para poder ejecutar la operación. En **C±** la realización o implementación será la definición completa del subprograma, en forma de *bloque de código*.

Tomemos como ejemplo una función que calcule el máximo de dos números:

```
                                             -----------------------------.
int Maximo2( int a, int b ) {                 Especificación: Sintaxis   |
                                             -----------------------------+
/* Maximo2(a, b) es el Máximo de a y b */    Especificación: Semántica  |
                                             -----------------------------+
  if (a > b) {                                                           |
    return a;                                                            |
  } else {                                    Realización                |
    return b;                                                            |
  }                                                                      |
}                                            -----------------------------'
```

Conociendo sólo la especificación podemos invocar la función, aunque no sepamos el detalle de cómo se realiza. Por ejemplo, podemos escribir:

```
alturaTotal = Maximo2( altura1, altura2 );
```

Si ahora sustituimos la realización de la función **Maximo2** por otra diferente, tal como:

```
int Maximo2( int a, int b ) {
  int m;

  m = a;
  if (b > m) {
    m = b;
  }
  return m;
}
```

la especificación de la función sigue siendo la misma. La sentencia anterior que usaba la función sigue siendo correcta:

```
alturaTotal = Maximo2( altura1, altura2 );
```

Con ello se pone de manifiesto la idea de que la especificación es una visión abstracta de qué hace la función, con independencia de los detalles de cómo lo hace. Precisamente las reglas de visibilidad de **C±** permiten usar subprogramas como operaciones abstractas, con *ocultación* de los detalles de realización.

Es importante comprender que si describimos la semántica en lenguaje humano, impreciso, tendremos sólo una *especificación informal.* Si se necesita mayor rigor se puede recurrir a expresiones lógico-matemáticas para *especificar formalmente* las condiciones que relacionan los datos de entrada y los resultados. La especificación formal evita ambigüedades, pero también suele resultar más costosa de escribir y más difícil de leer. Por ello conviene que vaya acompañada siempre de una especificación en lenguaje humano. En el apéndice C se describe la notación utilizada en este libro para las especificaciones formales.

La especificación formal del máximo de dos valores podría ser:

$$Maximo2(a, b) = (a \geq b \Rightarrow a|b)$$

En esta especificación se ha usado una expresión condicional. El primer ejemplo de código de esta sección usa como realización una transcripción directa de esta especificación formal. Podemos repetir ahora dicho ejemplo anotándolo con la PRECONDICIÓN y POSCONDICIÓN correspondientes a su especificación formal :

```
int Maximo2( int a, int b ) {
  «PRE : »
  «POST : Maximo2(a, b) es el Máximo de a y b »
  «POST : Maximo2(a,b) = (a ≥ b ⇒ a|b) »
```

```
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

La PRECONDICIÓN está vacía (se asume que siempre es cierta) porque no hay restricciones en los valores aceptables de los argumentos. El valor del máximo está definido para toda pareja de valores enteros. La POSTCONDICIÓN se ha escrito dos veces, primero en lenguaje natural, informal, y luego de manera formal.

### 8.1.2 Funciones. Argumentos

En programación la idea de función surge al aplicar el concepto de *abstracción* a las expresiones aritméticas. Una expresión representa un nuevo valor obtenido por cálculo a partir de ciertos valores ya conocidos que se usan como operandos.

Por ejemplo, el cubo de un número $Z$ se puede calcular multiplicando el número por sí mismo, de la forma $Z \times Z \times Z$. De esta manera se puede obtener, por ejemplo, el volumen de un cubo escribiendo:

```
volumen = lado * lado * lado;
```

La expresión `lado * lado * lado` suministra el valor del cubo del `lado` al evaluarla. Esta expresión puede verse de manera *abstracta* como una función, siendo `lado` el dato de partida y el cubo el resultado obtenido. La abstracción de dicha expresión tendrá asociado un nombre que podamos identificar con el significado del cálculo, y que, obviamente, podría ser `Cubo`. Esto nos conduciría a la especificación:

```
                                 ------------------------------.
float Cubo( float z )               Especificación: Sintaxis   |
                                 ------------------------------+
/* Cubo(z) = z³ */                  Especificación: Semántica  |
                                 ------------------------------'
```

O bien, de manera formal:

```
float Cubo( float z )
  «PRE: »
  «POST: Cubo(z) = z³ »
```

Con esta especificación, el cálculo del volumen se reescribiría como:

```
volumen = Cubo( lado )
```

Esta visión abstracta prescinde de los detalles de cómo se calcula el resultado, con tal de que sea correcto, es decir, que se obtenga el cubo del argumento. La realización puede ser tan sencilla como:

```
float Cubo( float z ) {
  return z * z * z;
}
```

o tan artificiosa como:

```
float Cubo( float z ) {
  float c = 1.0;

  for (int k = 1; k <= 3; k++) {
    c = c * z;
  }
  return c;
}
```

Los operandos que intervienen en el cálculo del valor de la función y que pueden cambiar de una vez a otra se especifican como argumentos de dicha función. La función aparece así como una *expresión parametrizada*.

En el tema anterior se han mencionado ya las dos formas disponibles en **C±** para el paso de los argumentos al subprograma que realiza el cálculo de la función. Si buscamos que el concepto de función en programación se aproxime al concepto matemático de función, el paso de argumentos debería ser siempre por valor. El concepto matemático de función es una aplicación entre conjuntos, cuyo cómputo se limita a suministrar un resultado, sin modificar el valor de los argumentos.

Aunque algunas veces, por razones de eficiencia, pueda ser aconsejable pasar por referencia argumentos de funciones, seguirá siendo deseable, para mantener la máxima claridad en el programa, que la llamada a la función no modifique el valor de los argumentos.

Desde el punto de vista de claridad del programa, y con independencia de cuál sea el mecanismo de paso de agumentos empleado, la cualidad más deseable al utilizar funciones es conseguir su *transparencia referencial*. Tal como se mencionó anteriormente, la transparencia referencial significa que la función devolverá siempre el mismo resultado cada vez que se la invoque con los mismos argumentos.

La transparencia referencial se garantiza si la realización de la función no utiliza datos exteriores a ella. Es decir, si no emplea:

- Variables externas al subprograma, a las que se accede directamente por su nombre, de acuerdo con las reglas de visibilidad de bloques.
- Datos procedentes del exterior, obtenidos con sentencias de lectura.
- Llamadas a otras funciones o procedimientos que no posean transparencia referencial. Las sentencias de lectura son en realidad un caso particular de éste.

Estas restricciones se cumplen en el ejemplo anterior del cálculo del cubo de un número. Las funciones que cumplen la cualidad de transparencia referencial y que no producen efectos laterales o secundarios se denominan *funciones puras*.

### 8.1.3 Acciones abstractas. Procedimientos

De manera similar a como las funciones pueden ser consideradas como expresiones abstractas, parametrizadas, los procedimientos pueden ser considerados como *acciones abstractas*, igualmente *parametrizadas*. Un procedimiento representa una acción, que se define por separado, y que se invoca por su nombre.

Como acciones *abstractas*, podemos tener dos visiones de un procedimiento. La primera es la visión abstracta o *especificación*, formada por la cabecera del procedimiento y una descripción de *qué hace* dicho procedimiento, y la segunda es la *realización*, en que se detalla, codificada en el lenguaje de programación elegido, *cómo se hace* la acción definida como procedimiento.

Como ejemplo, definiremos la acción abstracta de intercambiar los valores de dos variables. La especificación podría ser:

```
                                        ----------------------------.
void Intercambiar( int & a, int & b )    Especificación: Sintaxis   |
                                        ----------------------------+
/* (a',b') = (b,a) */                    Especificación: Semántica  |
                                        ----------------------------'
```

Esta especificación es realmente formal, ya que equivale exactamente a:

```
void Intercambiar( int & a, int & b )
  «PRE : »
  «POST : (a', b') = (b,a) »
```

Para escribir esta especificación hemos necesitado distinguir los valores de los argumentos (pasados por referencia) en dos momentos diferentes: al comienzo y al final de la ejecución del procedimiento. Los nombres con prima (′) representan los valores finales. La expresión:

$$(a',\ b') = (b, a)$$

significa que la pareja de valores de los argumentos $a$ y $b$, por este orden, al terminar la ejecución del subprograma, coincide con la pareja de valores de $b$ y $a$, por este orden, al comienzo de la ejecución del subprograma.

Conociendo la especificación podemos ya escribir algún fragmento de programa que utilice este procedimiento. Si queremos ordenar dos valores de menor a mayor, podríamos escribir:

```
if (p > q) {
  Intercambiar( p, q );
}
```

Para escribir este fragmento de programa no hemos necesitado saber cuál es la realización del procedimiento de intercambiar. Por supuesto, para tener un programa completo, que se pueda ejecutar, necesitamos escribir una realización válida. Por ejemplo:

```
void Intercambiar( int & a, int & b ) {
  int aux;

  aux = a;
  a = b;
  b = aux;
}
```

Al definir procedimientos no podemos limitarnos a usar sólo el paso de argumentos por valor. En programación imperativa las acciones consisten habitualmente en modificar los valores de determinadas variables. Por esta razón se considera normal que los procedimientos usen argumentos pasados por referencia.

De todas maneras conviene seguir una cierta disciplina para que los programas resulten claros y fáciles de entender. Para ello podemos recomendar que los procedimientos se escriban siempre como *procedimientos puros*, entendiendo por ello que no produzcan efectos laterales o secundarios. Con esto se consigue que la acción que realiza un procedimiento se deduzca en forma inmediata de la invocación de dicha acción. Se garantiza que un procedimiento cumple con esta cualidad si su realización no utliza:

- Variables externas al subprograma, a las que se accede directamente por su nombre, de acuerdo con las reglas de visibilidad de bloques.
- Llamadas a otros subprogramas que no sean procedimientos o funciones puras.

Comparando esta lista de restricciones con la que se estableció para las funciones puras, se observa que hemos suprimido la exigencia de que el procedimiento

no lea datos del exterior. En general esta lectura puede considerarse como una asignación de valor, que puede quedar suficientemente bien reflejada en la llamada, si los dispositivos o ficheros de entrada se mencionan explícitamente como argumentos.

De todas maneras es difícil establecer una disciplina precisa con recomendaciones sobre la definición y uso de procedimientos. Hay muchas situaciones en las que la claridad del programa aumenta, de hecho, si se usan procedimientos en los que se accede a variables globales. Así es posible evitar que haya que escribir repetidamente argumentos iguales en cada una de las llamadas al procedimiento. En particular, algunos de los procedimientos de lectura (o de escritura) del módulo `stdio` omiten pasar como argumento el fichero de datos de entrada (o de salida), y asumen por defecto una entrada y una salida principales de datos, predefinidas (teclado y pantalla, respectivamente).

## 8.2 Desarrollo usando abstracciones

La metodología de programación estructurada puede ampliarse con la posibilidad de definir operaciones abstractas mediante subprogramas. A continuación se describen dos estrategias de desarrollo diferentes, según qué se escriba primero, si la definición de los subprogramas, o el programa principal que los utiliza.

### 8.2.1 Desarrollo descendente

La estrategia de *desarrollo descendente* (en inglés, *Top-Down*), es simplemente el desarrollo por refinamientos sucesivos, teniendo en cuenta además la posibilidad de definir operaciones abstractas. En cada etapa de refinamiento de una operación habrá que optar por una de las alternativas siguientes:

- Considerar la operación como *operación terminal*, y codificarla mediante sentencias del lenguaje de programación.
- Considerar la operación como *operación compleja*, y descomponerla en otras más sencillas.
- Considerar la operación como *operación abstracta*, y especificarla, escribiendo más adelante el subprograma que la realiza.

Para decidir si una operación debe refinarse como operación abstracta habrá que analizar las ventajas que se obtengan, frente a la codificación directa o descomposición de la operación en forma de un esquema desarrollado en ese punto del programa.

En general resultará ventajoso refinar una operación como operación abstracta, que se define en forma separada, si se consigue alguna de las ventajas siguientes.

- Evitar mezclar en un determinado fragmento de programa operaciones con un nivel de detalle muy diferente.
- Evitar escribir repetidamente fragmentos de código que realicen operaciones análogas.

El beneficio obtenido es, como cabría esperar, una mejora en la claridad del programa. Hay que decir que esto implica un costo ligeramente mayor en términos de eficiencia, ya que siempre se ejecuta más rápidamente una operación si se escriben directamente las sentencias que la realizan, que si se invoca un subprograma que contiene dichas sentencias. La llamada al subprograma representa una acción adicional que consume un cierto tiempo de ejecución.

Por el contrario, hay un aumento de eficiencia en ocupación de memoria si se codifica como subprograma una operación que se invoca varias veces en distintos puntos del programa. En este caso el código de la operación aparece sólo una vez, mientras que si se escribiesen cada vez las sentencias equivalentes el código aparecería repetido varias veces.

## 8.2.2 Ejemplo: Imprimir la figura de un árbol de navidad

Retomamos aquí el ejemplo desarrollado en el tema 4. El objetivo es imprimir la silueta del árbol, tal como aparece a continuación:

```
     *
    ***
   *****
    ***
   *****
  *******
   *****
  *******
 *********
     *
     *
     *
   *****
```

Los primeros pasos de refinamiento eran:

*Imprimir árbol* ⟶
*Imprimir copa*
*Imprimir tronco*
*Imprimir base*

*Imprimir copa* ⟶
*Imprimir primeras ramas*
*Imprimir segundas ramas*
*Imprimir terceras ramas*

Podemos observar la existencia de operaciones análogas, correspondientes a la impresión de los distintos fragmentos. Es relativamente sencillo darse cuenta de que cada una de las "ramas" de la copa del árbol es una figura trapezoidal. Por ejemplo, las "segundas ramas" aparecen dibujadas así:

```
  ***
 *****
*******
```

Esta figura geométrica es un trapecio simétrico. Lo mismo puede decirse de las otras "ramas". Puesto que cada vez se imprime una figura diferente, podremos definir esta acción como parametrizada, dando como argumentos la información necesaria para distinguir cada "rama" particular. Por ejemplo, podemos decidir que el único parámetro necesario es la anchura de la base superior, ya que todas las "ramas" tienen 3 líneas de altura, y cada una de estas líneas añade siempre un asterisco más a cada lado.

La especificación la impresión de una "rama" como procedimiento se podrá redactar de la forma:

```
/*============================================
  Especificación Semántica: Procedimiento para
  imprimir 3 líneas seguidas con:
         ancho,
         ancho + 2
       y ancho + 4 asteriscos
  ============================================*/

void ImprimirRama( int ancho )        /* Especificación Sintáctica  */
```

En cuanto a la impresión del tronco y la base, también cabe la posibilidad de considerarlas como operaciones análogas, en ambos casos un rectángulo de asteriscos. Los parámetros serían en este caso la anchura y altura del rectángulo. La especificación sería:

```
/*==============================================
  Especificación Semántica: Procedimiento para
  imprimir un rectángulo de ancho × alto asteriscos
  ==============================================*/

void ImprimirRectangulo( int ancho, int alto )
/* Especificación Sintáctica */
```

Con esto se podría escribir ya el programa principal, en el que podemos agrupar la impresión de las ramas en un esquema de bucle.

```
/*-- Imprimir copa --*/
rama = 1;
for (int k = 1; k <= 3; k++) {
  ImprimirRama( rama );
  rama = rama + 2;
}

/*-- Imprimir tronco --*/
ImprimirRectangulo( 1, 3 );

/*-- Imprimir base --*/
ImprimirRectangulo( 5, 1 );
```

Ahora falta escribir la realización de las operaciones abstractas especificadas anteriormente. Forzando quizá un poco la idea de buscar operaciones análogas, se puede establecer una relación entre la impresión de las "ramas" y las del tronco o la base. En efecto, un rectángulo puede considerarse como un caso particular de un trapecio. Tanto la operación de **ImprimirRama** como la de `ImprimirRectangulo` se pueden apoyar en una operación común de `ImprimirTrapecio` especificada de la forma siguiente:

```
/*================================================
  Procedimiento para imprimir un trapecio de asteriscos
  con la base superior "ancho", altura "alto" y "avance"
  asteriscos más a cada lado en cada nueva línea
  ================================================*/

void ImprimirTrapecio( int ancho, int alto, int avance )
```

Esta operación la desarrollaremos mediantes refinamientos:

*Imprimir trapecio* ⟶

```
for (int k = 1;k <= alto; k++) {
  Imprimir una línea del trapecio
}
```

*Imprimir una línea del trapecio* $\longrightarrow$
*Imprimir los blancos iniciales*
*Imprimir los asteriscos*

Para mantener la información del número de asteriscos en cada línea usaremos una variable **anchura**, que tomará inicialmente el ancho de la línea superior, y se irá incrementando después de imprimir cada línea. Los blancos iniciales se calculan cada vez, fijando como parámetro constante la posición del centro de la línea.

Al desarrollar este procedimiento se observa una analogía entre la operación de escribir los espacios en blanco y la de escribir los asteriscos. Ambas operaciones se refinan como la operación abstracta de imprimir un mismo carácter un cierto número de veces. Para ello se especifica el procedimiento:

```
/*============================================================
  Procedimiento para imprimir N veces seguidas el carácter 'c'
  ============================================================*/

void ImprimirN( char c, int N )
```

El programa completo, incluyendo todos los procedimientos, es el siguiente:

```
/*****************************************************
* Programa: ArbolDeNavidad
*
* Descripción:
*   Este programa imprime la silueta de un árbol
*   de Navidad, hecha con asteriscos.
*****************************************************/
#include <stdio.h>

/*=================
  Constante global
  =================*/
const int centro = 20;   /* Centro de cada línea */

/*============================================================
  Procedimiento para imprimir N veces seguidas el carácter 'c'
  ============================================================*/
void ImprimirN( char c, int N ) {
  for (int k =1; k <= N; k++) {
    printf( "%c", c );
  }
}
```

```
/*==========================================================
  Procedimiento para imprimir un trapecio de asteriscos
  con la base superior "ancho", altura "alto" y "avance"
  asteriscos más a cada lado en cada nueva línea
  ==========================================================*/
void ImprimirTrapecio( int ancho, int alto, int avance ) {
  int anchura;       /* número de asteriscos */

  anchura = ancho;
  for (int k = 1; k <= alto; k++) {
    ImprimirN( ' ', centro - anchura/2 );
    ImprimirN( '*', anchura );
    printf( "\n" );
    anchura = anchura + 2*avance;
  }
}

/*==============================================
  Procedimiento para imprimir 3 líneas seguidas
  con:   ancho
        ancho+2
      y ancho+4   asteriscos
  ==============================================*/
void ImprimirRama( int ancho ) {
  ImprimirTrapecio( ancho, 3, 1 );
}

/*==========================================
  Procedimiento para imprimir un rectángulo
  de ancho x alto asteriscos
  ==========================================*/
void ImprimirRectangulo( int ancho, int alto ) {
  ImprimirTrapecio( ancho, alto, 0 );
}

/*==========================================
  Programa principal
  ==========================================*/
int main() {
  int rama;          /* Ancho de rama */

  /*-- Imprimir copa --*/
  rama = 1;
  for (int k = 1; k <= 3; k++) {
    ImprimirRama( rama );
    rama = rama + 2;
```

```
  }

  /*-- Imprimir tronco --*/
  ImprimirRectangulo( 1, 3 );

  /*-- Imprimir base --*/
  ImprimirRectangulo( 5, 1 );
}
```

Comparando esta redacción del programa con la que se había desarrollado en el tema 4, se observa que el programa resulta ahora más largo, aunque cada parte separada del programa es más sencilla. En la versión anterior la parte ejecutable del programa principal era más compleja que ahora.

Con esta nueva redacción se obtiene una ventaja adicional, que se ha producido como efecto de la labor de abstracción realizada para especificar los subprogramas. Operaciones que antes se consideraban por separado, ahora se han refundido en una sola operación abstracta y parametrizada. La parametrización tiene la ventaja de que se facilita la modificación posterior del programa.

En efecto, si quisiéramos cambiar el programa para imprimir una figura de árbol algo diferente, en la versión inicial habría sido necesario cambiar casi toda la parte ejecutable del programa, sentencia por sentencia. Ahora la mayor parte del programa está constituida por las definiciones de las operaciones abstractas, que se pueden mantener sin cambios, y sólo hay que rectificar la parte de código del programa principal, que es relativamente corta.

Por ejemplo, podemos modificar el código del programa principal para imprimir un árbol más grande, tal como se indica en el programa **ArbolGrande**, donde los cambios se han destacado con un recuadro [x].

Con la versión inicial del programa habríamos tenido que escribir de nuevo al menos unas 13 líneas del programa. Ahora no ha sido necesaria ninguna línea nueva y tan sólo hemos tenido que retocar 3, y sólo para modificar los valores de los tamaños.

```
/***************************************************
*    Programa: ArbolGrande de Navidad
***************************************************/
/* Usa las mismas definiciones de procedimientos
   que el programa anterior */
```

```
int main() {
  int rama;           /* Ancho de rama */

  /*-- Imprimir copa más alta--*/
  rama = 1;
  for (int k = 1; k <= 5; k++) {
    ImprimirRama(rama);
    rama = rama + 2;
  }

  /*-- Imprimir tronco más grueso y alto --*/
  ImprimirRectangulo(3, 5);

  /*-- Imprimir base más ancha y alta --*/
  ImprimirRectangulo(9, 2);
}
```

El resultado de la ejecución del programa ArbolGrande es el siguiente:

```
                    *
                   ***
                  *****
                   ***
                  *****
                 *******
                  *****
                 *******
                *********
                 *******
                *********
               ***********
                *********
               ***********
              *************
                   ***
                   ***
                   ***
                   ***
                   ***
                *********
                *********
```

### 8.2.3 Ejemplo: Imprimir una tabla de números primos

En el ejemplo anterior se buscó de manera insistente la analogía entre operaciones, y su especificación como operaciones parametrizadas. En este ejemplo se atenderá fundamentalmente a la limitación en el nivel de detalle.

El objetivo de este programa de ejemplo es imprimir una tabla con los números primos hasta un límite dado, formando varias columnas de números a lo ancho del listado. Si decidimos imprimir los números primos hasta 100, a cuatro columnas de 15 caracteres de ancho cada una, el resultado deberá ser el que aparece a continuación:

```
              1              2              3              5
              7             11             13             17
             19             23             29             31
             37             41             43             47
             53             59             61             67
             71             73             79             83
             89             97
```

Los primeros pasos de refinamiento serán:

*Imprimir la tabla de números primos de 1 a N* $\longrightarrow$

```
for (int k = 1; k <= N; k++) {
  Imprimir k, si es primo
}
```

*Imprimir k, si es primo* $\longrightarrow$

```
if ( k es primo ) {
  Imprimir k, tabulando
}
```

Ahora decidimos limitar el nivel de detalle, y definir como operaciones abstractas las que faltan por refinar. Sus especificaciones serían:

```
/*================================================
  Función para ver si un valor "k" es un número primo
  ================================================*/
bool EsPrimo( int k )
  «PRE: k > 0 »
  «POST: k no tiene divisores distintos de él mismo y la unidad »
  «POST: EsPrimo(k) = ∀d ∈ (2..k − 1) • modulo(k, d) ≠ 0 »
```

```
/*==================================
  Procedimiento para imprimir tabulando
  a 4 columnas de 15 caracteres
  ==================================*/
void ImprimirTabulando( int k )
```

NOTA: El tipo predefinido `bool`, que se introducirá en el próximo tema, sólo puede tomar dos valores posibles: `true` y `false`, que son los denominados hasta ahora SI/NO como posible resultado de una condición.

La función `EsPrimo` se especifica formalmente (y también informalmente, para facilitar el comprender la expresión formal). Se omite la especificación formal del procedimiento `ImprimirTabulando`, ya que es más difícil de redactar. De hecho es difícil formalizar las operaciones de lectura y escritura, en general, ya que exigiría especificar formalmente la representación externa de los datos y resultados.

A continuación podemos desarrollar la realización de estos subprogramas. La función que determina si un número es primo puede realizarse sencillamente desarrollando su especificación formal, es decir, probando todos los divisores posibles. Esta realización es poco eficiente, pero muy sencilla de programar. La única optimización es que en cuanto se encuentra un divisor `d` ya no se prueban otros, puesto que eso garantiza que el predicado $\forall d \in ...$ no puede cumplirse.

```
/*============================================
  Función para ver si un valor "k" es un número primo
  ============================================*/
bool EsPrimo( int k ) {

  /* d es un posible divisor */
  for (int d = 2; d <= k-1; d++) {
    if ((k % d) == 0) {
      return false;
    }
  }
  return true;
}
```

Para desarrollar la realización del procedimiento de imprimir tabulando hay que analizar algunas cuestiones previas. La especificación se ha establecido pasando como argumento solamente el número que hay que imprimir, reflejando de esta manera la forma natural en que se ha descrito esta acción abstracta. Sin embargo esta información es insuficiente para realizar la acción, ya que

es necesario saber qué columna toca imprimir para poder decidir si hay que saltar de línea o no.

En este ejemplo se decide usar una variable global **columna** para mantener dicha información. La variable contendrá en cada momento el número de la columna en que aparecería escrito el próximo número si previamente no se saltase de línea.

El refinamiento de esta operación será el siguiente:

*Imprimir k , tabulando* ⟶
  *Saltar de línea, si es necesario*
  *Imprimir k y actualizar la columna*

*Saltar de línea, si es necesario* ⟶

```
if (columna > 4) {
  columna = 1;
  printf( "\n" );
}
```

El programa completo, incluyendo la definición de todos los subprogramas necesarios, es el siguiente:

```
/****************************************************
* Programa: Primos
*
* Descripción:
*   Este programa imprime una tabla de números
*   primos, tabulando a cuatro columnas
****************************************************/
#include <stdio.h>

/*=================
  Constante global
  =================*/
const int N = 100;      /* rango de números */

/*===============
  Variable global
=================*/
int columna;            /* columna a imprimir */
```

```
/*==========================================================
  Función para ver si un valor "k" es un número primo
  ==========================================================*/
bool EsPrimo( int k ) {
  for (int d = 2; d <= k-1; d++) {
    if ((k % d) == 0) {
      return false;
    }
  }
  return true;
}

/*========================================
  Procedimiento para imprimir tabulando
  a 4 columnas de 15 caracteres
  ========================================*/
void ImprimirTabulando( int k ) {
  if (columna > 4) {
    columna = 1;
    printf( "\n" );
  }
  printf( "%15d", k );
  columna++;
}

/*==================
  Programa principal
  ==================*/
int main() {
  columna = 1;
  for (int k = 1; k <= N; k++) {
    if (EsPrimo(k)) {
      ImprimirTabulando( k );
    }
  }
  printf( "\n" );
}
```

### 8.2.4 Reutilización

La realización de ciertas operaciones como subprogramas independientes facilita lo que se llama *reutilización* de software. Si la operación identificada como operación abstracta tiene un cierto sentido en sí misma, es muy posible que resulte de utilidad en otros programas, además de en aquél para el cual

se ha desarrollado. La escritura de otros programas que utilicen esa misma operación resulta más sencilla, ya que se aprovecha el código de su definición, que ya estaba escrito.

Aplicaremos esta idea a los subprogramas desarrollados para imprimir el árbol de Navidad. Las operaciones abstractas definidas allí permiten imprimir con bloques de asteriscos figuras trapezoidales, o simplemente rectangulares, de dimensiones variables. Cualquier figura que pueda descomponerse en secciones de estas formas se podrá imprimir fácilmente usando los procedimientos ya definidos.

Por ejemplo, podremos imprimir la figura de una casa de juguete, tal como la siguiente:

```
    **
    **
*********
***********
*************
*********
*********
*********
```

Para ello sólo tendremos que escribir un fragmento de programa así:

```
int main() {
  /*-- Imprimir chimenea--*/
  ImprimirRectangulo( 2, 2 );

  /*-- Imprimir tejado --*/
  ImprimirRama( 9 );

  /*-- Imprimir cuerpo de la casa--*/
  ImprimirRectangulo( 9, 3 );
}
```

Por supuesto, tendremos que copiar en la parte de declaraciones las definiciones de los procedimientos ya desarrollados en el programa del árbol de Navidad.

A continuación se presentan más ejemplos, que aprovechan subprogramas desarrollados de antemano.

### 8.2.5 Ejemplo: Tabular la serie de Fibonacci

El procedimiento de imprimir tabulando desarrollado en el ejemplo de imprimir la tabla de números primos, puede aprovecharse para imprimir en forma

de tabla otras series de valores. Por ejemplo, podemos tabular la serie de Fibonacci, que ya se describió en el tema 6. Lo que necesitamos ahora es sustituir las sentencias de escritura usadas en aquel ejemplo

```
printf( "%10d\n", termino );
```

por una llamada al procedimiento

```
ImprimirTabulando( termino );
```

Sólo falta copiar en la parte de declaraciones la definición del procedimiento de tabular y añadir al comienzo del programa la inicialización del contador de columnas. El programa completo aparece listado a continuación:

```
/*************************************************
* Programa: Fibonacci
*
* Descripción:
*   Este programa imprime todos los términos
*   de la serie de Fibonacci dentro del rango de
*   valores positivos del tipo int: (1 .. INT_MAX)
*   Se imprime tabulando a cuatro columnas
*************************************************/
#include <stdio.h>
#include <limits.h>

/*================
  Variable global
=================*/
int columna;          /* columna a imprimir */

/*====================================
  Procedimiento para imprimir tabulando
  a 4 columnas de 15 caracteres
  ====================================*/
void ImprimirTabulando( int k ) {
  if (columna > 4) {
    columna = 1;
    printf( "\n" );
  }
  printf( "%15d", k );
  columna++;
}
```

```
/*==================
  Programa principal
  ==================*/
int main() {
  int termino;               /* término de la serie */
  int anterior;              /* término anterior */
  int aux;

  /*-- Iniciar la tabulación --*/
  columna = 1;

  /*-- Generar el comienzo de la serie --*/
  anterior = 0;
  termino = 1;
  ImprimirTabulando( anterior );
  ImprimirTabulando( termino );

  /*-- Generar el resto de la serie --*/
  while (INT_MAX - termino >= anterior) {
    aux = termino + anterior;
    anterior = termino;
    termino = aux;
    ImprimirTabulando( termino );
  }
  printf( "\n" );
}
```

El resultado de la ejecución en una máquina con números enteros de 32 bits (INT_MAX = 2.147.483.647) es el siguiente:

```
          0          1          1          2
          3          5          8         13
         21         34         55         89
        144        233        377        610
        987       1597       2584       4181
       6765      10946      17711      28657
      46368      75025     121393     196418
     317811     514229     832040    1346269
    2178309    3524578    5702887    9227465
   14930352   24157817   39088169   63245986
  102334155  165580141  267914296  433494437
  701408733 1134903170 1836311903
```

### 8.2.6 Desarrollo para reutilización

Para aplicar de manera eficaz las técnicas de reutilización de software es preciso pensar en las posibles aplicaciones de un cierto subprograma en el momento de especificarlo, con independencia de las necesidades particulares del programa que se está desarrollando en ese momento.

Esta estrategia de desarrollo tiene ventajas e inconvenientes. La principal ventaja es que se amplía el conjunto de aplicaciones en que se podrá reutilizar más adelante el subprograma que se está desarrollando ahora. Su principal inconveniente es que será más costoso hacer el desarrollo del subprograma planteado como operación de uso general, que planteado como operación particular, hecha a medida del programa que lo utiliza en este momento.

En el ejemplo del árbol de Navidad, nos encontramos con que al buscar analogías entre distintas operaciones para resolverlas con un subprograma común, estábamos generalizando al mismo tiempo dichas operaciones, estableciendo parámetros que permitían particularizarla para cada caso.

En el caso de subprogramas planteados simplemente con el fin de limitar el nivel de detalle en una sección determinada de un programa, no se siente esta necesidad de generalizar, y es más facil plantear la operación particularizada para las necesidades de ese momento.

En el ejemplo de tabular las series de valores, se ha planteado de entrada la operación de tabulación de manera que impone tanto el número de columnas como el ancho de cada una. Si queremos escribir un subprograma de tabulación de resultados que sea realmente de uso general, convendría dejar libertad para fijar las características del listado como parámetros modificables, que se puedan particularizar para cada caso.

De esta manera se podría haber ampliado el campo de aplicación del subprograma de tabular si el número de columnas y el ancho de cada una fuesen parámetros variables. Además, para simplificar el uso del procedimiento de tabulación se podrían agrupar todas las acciones de inicialización en una sola acción abstracta, invocada como subprograma, en que se fijen los parámetros particulares del listado. La especificación de esta acción inicial podría ser:

```
/*======================================
  Procedimiento para iniciar la tabulación
  con los parámetros indicados
  ======================================*/
void IniciarTabulacion( int columnas, int ancho )
```

Para ilustrar esta técnica, modificaremos el programa de tabular la serie de Fibonacci de acuerdo con lo expuesto, decidiendo el formato del listado (6 co-

lumnas de 11 caracteres) desde el programa principal. El programa modificado es el siguiente:

```
/***************************************************
* Programa: Fibonacci2
*
* Descripción:
*   Este programa imprime todos los términos
*   de la serie de Fibonaci, dentro del rango de
*   valores positivos del tipo int: (1 .. INT_MAX)
*   Se imprime tabulando a siete columnas
***************************************************/
#include <stdio.h>
#include <limits.h>

/*=================
  Variables globales
====================*/
int TABcolumna;          /* columna a imprimir */
int TABultima;           /* última columna */
int TABancho;            /* ancho de cada columna */

/*=====================================
  Procedimiento para iniciar la tabulación
  con los parámetros indicados
  ======================================*/
void IniciarTabulacion( int columnas, int ancho ) {
  TABultima  = columnas;
  TABancho   = ancho;
  TABcolumna = 1;
}

/*====================================
  Procedimiento para imprimir tabulando
  a TABultima columnas
  de TABancho caracteres
  =====================================*/
void ImprimirTabulando( int k ) {
  if (TABcolumna > TABultima) {
    TABcolumna = 1;
    printf( "\n" );
  }
  printf( "%*d",TABancho, k );
  TABcolumna++;
}
```

```
/*==================
  Programa principal
  ==================*/
int main() {
  int termino;                /* término de la serie */
  int anterior;               /* término anterior */
  int aux;

  /*-- Iniciar la tabulación --*/
  IniciarTabulacion( 6, 11 );

  /*-- Generar el comienzo de la serie --*/
  anterior = 0;
  termino = 1;
  ImprimirTabulando( anterior );
  ImprimirTabulando( termino );

  /*-- Generar el resto de la serie --*/
  while (INT_MAX - termino >= anterior) {
    aux = termino + anterior;
    anterior = termino;
    termino = aux;
    ImprimirTabulando( termino );
  }
  printf( "\n" );
}
```

El resultado de la ejecución es el siguiente:

```
          0          1          1          2          3          5
          8         13         21         34         55         89
        144        233        377        610        987       1597
       2584       4181       6765      10946      17711      28657
      46368      75025     121393     196418     317811     514229
     832040    1346269    2178309    3524578    5702887    9227465
   14930352   24157817   39088169   63245986  102334155  165580141
  267914296  433494437  701408733 1134903170 1836311903
```

Conviene comentar algunos aspectos de estilo utilizados en este ejemplo. Las variables globales para la tabulación se han nombrado empezando sus nombres con el prefijo TAB, para establecer que están todas ellas relacionadas y que son visibles desde todos los subprogramas. Sin embargo, las variables para el cálculo de la serie de Fibonacci están declaradas como locales al programa principal y se pasan como argumentos al procedimiento de **ImprimirTabulan-**

do separando así el cálculo de la serie y la reutilización de la impresión de un número cualquiera tabulando.

En realidad esto es un recurso artificioso para separar las distintas partes del programa. Este recurso se utilizará más ampliamente y de manera natural cuando se presenten los *tipos abstractos de datos* mediante módulos separados en **C±**. Usando el mecanismo de *módulos* se pueden desarrollar subprogramas reutilizables, escritos en forma realmente independiente, de una manera mucho más sencilla y adecuada.

### 8.2.7 Desarrollo ascendente

La metodología de *desarrollo ascendente* (en inglés *Bottom-Up*) consiste en ir creando subprogramas que realicen operaciones significativas de utilidad para el programa que se intenta construir, hasta que finalmente sea posible escribir el programa principal, de manera relativamente sencilla, apoyándose en los subprogramas desarrollados hasta ese momento.

La técnica tiene una cierta analogía con el desarrollo de subprogramas pensando en su reutilización posterior. Al hablar de desarrollo para reutilización se ha dicho que los subprogramas podían surgir en el proceso de refinamiento de un programa concreto, al identificar ciertas operaciones, pero debían definirse pensando en futuras aplicaciones. En este caso se trata de que la identificación de las operaciones no surja de un proceso de descomposición o refinamiento de alguna acción en particular, sino simplemente pensando en el programa que se desarrolla, casi como una más de las posibles aplicaciones futuras.

Como ejemplo desarrollaremos un programa que opere como una calculadora, pero con fracciones. Una fracción se compondrá de un numerador y un denominador enteros. La calculadora podrá sumar, restar, multiplicar o dividir fracciones, y los resultados los presentará con la fracción simplificada, dividiendo por los factores comunes al numerador y al denominador.

Con independencia de los detalles de operación de la calculadora, pueden desarrollarse inicialmente procedimientos útiles para esta aplicación; en particular procedimientos para realizar cálculos con fracciones, así como leerlas o imprimirlas. En el siguiente listado se presenta una colección apropiada de procedimientos, sobre los cuales se podrá desarrollar luego el programa principal de la calculadora.

```
/*================================
  Procedimiento para simplificar
  la fracción n/d
  ================================*/
void ReducirFraccion( int & n, int & d ) {
  int divisor = 2;

  while ((divisor <= n) && (divisor <= d)) {
    while ((n % divisor == 0) && (d % divisor == 0)) {
      n = n / divisor;
      d = d / divisor;
    }
    divisor++;
  }
}

/*====================================
  Procedimiento para sumar fracciones
       n3´/d3´ = n1/d1 + n2/d2
  ====================================*/
void SumarFracciones( int n1, int d1, int n2, int d2,
                      int & n3, int & d3 ) {
  n3 = n1*d2 + n2*d1;
  d3 = d1*d2;
  ReducirFraccion( n3, d3 );
}

/*=====================================
  Procedimiento para restar fracciones
       n3´/d3´ = n1/d1 - n2/d2
  =====================================*/
void RestarFracciones( int n1, int d1, int n2, int d2,
                       int & n3, int & d3 ) {
  SumarFracciones( n1, d1, -n2, d2, n3, d3 );
}

/*=========================================
  Procedimiento para multiplicar fracciones
         n3´/d3´ = n1/d1 * n2/d2
  ====================================*/
void MultiplicarFracciones( int n1, int d1, int n2, int d2,
                            int & n3, int & d3 ) {
  n3 = n1*n2;
  d3 = d1*d2;
  ReducirFraccion( n3, d3 );
}
```

```
/*======================================
  Procedimiento para dividir fracciones
        n3´/d3´ = n1/d1 / n2/d2
  ======================================*/
void DividirFracciones( int n1, int d1, int n2, int d2,
                        int & n3, int & d3 ) {
  n3 = n1*d2;
  d3 = d1*n2;
  ReducirFraccion( n3, d3 );
}

/*===================================
  Procedimiento que lee una fracción
        y la simplifica
  ===================================*/
void LeerFraccion( int & n, int & d ) {
  scanf( "%d/%d", &n, &d );
  ReducirFraccion( n, d );
}

/*=========================
  Procedimiento que escribe
    una fracción como n/d
  =========================*/
void EscribirFraccion( int n, int d ) {
  printf( "%d/%d\n", n, d );
}
```

Contando con esos procedimientos se puede ahora desarrollar el programa principal de la calculadora, que se presenta en el programa `Fraccion`. En este ejemplo se supone que cada operación se realiza entre un valor acumulado y un nuevo operando. La operación se inicia con una tecla de operación, y a continuación se introduce el valor del operando. Las operaciones previstas son `+`, `-`, `*`, `/`. Además habrá teclas de operación para imprimir el resultado acumulado (=) y para iniciar una nueva serie de operaciones (`N`). La tecla `F` marcará el fin del funcionamiento del programa.

```
/*************************************************
* Programa: Fracciones
*
* Descripción:
*       Este programa es una calculadora que suma,
*       resta, multiplica y divide fracciones
*************************************************/
```

```
#include <stdio.h>

/*... definiciones de los procedimientos, omitidas ...*/

/*=================
  Programa principal
  =================*/
int main() {
  int num = 0;              /* Acumulador: Numerador */
  int den = 0;              /* Acumulador: Denominador */
  int nn, dd;               /* Nuevo operando a utilizar */
  char operacion = ' ';     /* Tecla de operación a realizar */

  while (operacion != 'F') {
    printf( ">> " );
    scanf( " %c", &operacion );

    if (operacion == '+') {
      LeerFraccion( nn, dd );
      SumarFracciones( num, den, nn, dd, num, den );

    } else if (operacion == '-') {
      LeerFraccion( nn, dd );
      RestarFracciones( num, den, nn, dd, num, den );

    } else if (operacion == '*') {
      LeerFraccion( nn, dd );
      MultiplicarFracciones( num, den, nn, dd, num, den );

    } else if (operacion == '/') {
      LeerFraccion( nn, dd );
      DividirFracciones( num, den, nn, dd, num, den );

    } else if (operacion == 'N') { /* Nuevos cálculos */
      LeerFraccion( num, den );

    } else if (operacion == '=') {
      printf( "         ") ;
      EscribirFraccion( num, den );

    } else if (operacion != 'F') {
      printf( "Pulse +, -, *, /, N, =, o F\n" );
    }
  }
}
```

Un posible ejemplo de la ejecución del programa es el siguiente:

```
>> N 5/20
>> =
          1/4
>> + 3/5
>> - 2/4
>> =
          7/20
>> * 5/6
>> =
          7/24
>> F
```

En esta aplicación de la técnica de desarrollo ascendente se puede apreciar que el desarrollo inicial de procedimientos para realizar cálculos con fracciones nos ha permitido disponer de una extensión del lenguaje **C±**, equivalente a definir el tipo FRACCION. Podríamos decir que los procedimientos de cálculo constituyen en conjunto una máquina virtual de operar con fracciones, sobre la cual se ha desarrollado el programa de la calculadora. El desarrollo es ascendente porque primero se han construído los subprogramas, de nivel inferior, y luego el programa que los usa, de nivel superior.

## 8.3 Programas robustos

La corrección de un programa exige que los resultados sean los esperados, siempre que el programa se ejecute con unos datos de entrada aceptables. La cuestión que nos ocupa en este momento es: ¿cuál debe ser el comportamiento del programa si los datos son incorrectos?.

Un programa se dice que es un *programa robusto* si su operación se mantiene en condiciones controladas aunque se le suministren datos erróneos.

### 8.3.1 Programación a la defensiva

La postura más cómoda desde el punto de vista del programador es declinar toda responsabilidad en el caso de que los datos no sean válidos. Si los datos de entrada no cumplen con los requisitos previstos, el programa puede entonces hacer cualquier cosa. Es frecuente que un programa se escriba sin tener en cuenta la posibilidad de que los datos no sean los esperados, pues con ello se simplifica su desarrollo.

Sin embargo esta postura no es admisible en la práctica. Como cualquier otra actividad humana, la escritura y uso de programas está sujeta a errores, y es importante conseguir que las consecuencias de esos errores no sean demasiado graves. Por ejemplo, un programa de gestión de un almacén deberá prever que se notifique la retirada de más cantidad de un producto que la anotada como existencias. En este caso el programa deberá hacer algo "razonable", tal como emitir un mensaje de aviso y obligar a repetir la operación, o simplemente asumir que el valor de las existencias estaba equivocado, y preguntar por el valor real de las existencias, o alguna otra cosa similar. Lo que no parece "razonable" es anotar un valor negativo para las existencias sin dar ningún aviso, o, en general, seguir operando con valores manifiestamente erróneos que podrían dar lugar más adelante a una parada indeseada del programa ("aborto") al intentar ejecutar alguna instrucción de máquina inadmisible con esos valores.

Otro ejemplo ilustrativo puede ser el de un programa para calcular el valor medio de una serie de datos, dividiendo la suma de todos por el número de datos introducidos. Cabe la posibilidad de que no se introduzca ningún dato, lo cual dará lugar a un intento de realizar una división por cero, que en muchos casos produce el "aborto" del programa. Si el cálculo de la media es lo único que hace el programa, el efecto no parece muy grave, pero si este cálculo es parte de las operaciones que realiza, por ejemplo, el programa de control de una central nuclear, los resultados pueden conducir a una catástrofe mundial. Lo realmente importante es detectar los errores en cuanto se produzcan, y poder así programar operaciones de corrección o tratamiento apropiadas para estas situaciones excepcionales.

La llamada *programación a la defensiva* (en inglés, *defensive programming*) consiste en que cada programa o subprograma esté escrito de manera que desconfíe sistemáticamente de los datos o argumentos con que se le invoca, y devuelva siempre como resultado:

(a) El resultado correcto, si los datos son admisibles, o bien
(b) Una indicación precisa de error, si los datos no son admisibles.

Lo que no debe hacer nunca el programa es devolver un resultado como si fuese normal, cuando en realidad es erróneo, ni "abortar". Esto da lugar a una propagación de errores, que puede aumentar la gravedad de las consecuencias, y hacer que la identificación del fallo del programa resulte mucho más difícil, ya que el efecto se puede manifestar sólo más adelante, en otra parte del programa sin relación aparente con la que falló.

La mejora de la robustez del programa tiene como contrapartida una cierta pérdida de eficiencia, al tener que hacer comprobaciones adicionales. Si la

eficiencia es un factor decisivo, algunas de estas comprobaciones pueden eliminarse en la versión final del programa, cuando se determine con seguridad que el programa no contiene errores.

Consideremos el caso de una función para calcular el factorial de un número:

$$n! = 1 \times 2 \times 3 \times ... \times n$$

La especificación de dicha función podría ser:

```
int Factorial( int n );
  «PRE : n ≥ 0 »
  «POST : Factorial(n) = (n = 0 ⇒ 1|n × Factorial(n − 1)»
```

Esta especificación nos indica que el factorial sólo está definido para valores de $n$ positivos, incluido cero, cuyo factorial por convenio vale $0! = 1$. El factorial de un número positivo no nulo se define a partir del anterior. Para valores negativos el factorial no está definido.

Una realización sencilla de la función podría ser:

```
int Factorial( int n ) {
  int f = 1;

  for (int k = 2; k <= n; k++) {
    f = f * k;
  }
  return f;
}
```

Esta función no es robusta. Para valores negativos el factorial no está definido, y sin embargo la función codificada de la forma anterior devuelve resultado 1, que sólo sería el resultado correcto correspondiente a 0! ó 1!.

Lo que hace falta es devolver una indicación clara de error para argumentos negativos. Una forma de hacerlo podría ser devolver un resultado cero o negativo en estos casos, ya que ese resultado no puede coincidir con el factorial de ningún número. La función se recodificaría como se indica a continuación. Como vemos, el código evalúa explícitamente la precondición. Si no se cumple devuelve un resultado fuera de rango, como indicación de error, y si se cumple devuelve el valor correcto:

```
int FactorialPositivo( int n ) {
  int f;
```

```
  if (n < 0) {
    f = 0;
  } else {
    f = 1;
    for (int k = 2; k <= n; k++) {
      f = f * k;
    }
  }
  return f;
}
```

En realidad la función sigue sin ser del todo robusta, ya que no se ha previsto la posibilidad de que el factorial que se intenta calcular exceda del rango admisible de valores del tipo `int`. Esto ocurre fácilmente incluso para valores relativamente pequeños del argumento (p.ej., 20! = 2.432.902.008.176.640.000). En la sección siguiente se presenta una versión más robusta de esta función.

### 8.3.2 Tratamiento de excepciones

Ante la posibilidad de errores en los datos con que se opera, hay que considerar dos actividades diferentes:

1. Detección de la situación de error.
2. Corrección de la situacion de error.

Si una operación se ha escrito como subprograma, la programación a la defensiva recomienda que la primera actividad (detección del posible error) se haga dentro del subprograma, sin confiar en que quienes usen el subprograma lo invoquen siempre con datos correctos.

Existen varios esquemas de programación posibles para tratamiento de errores. Un modelo recomendado es el *modelo de terminación*. En este modelo, si se detecta un error en una sección o bloque del programa, la acción de tratamiento del error reemplaza al resto de las acciones pendientes de dicha sección, con lo cual tras la acción correctora se da por terminado el bloque. En algunos lenguajes de programación, tales como el lenguaje Ada, Java y C++, existen construcciones o sentencias adecuadas para programar este esquema. El estudio profundo de las construcciones y mecanismos para tratamiento de excepciones quedan fuera de los objetivos de este libro.

También en **C±**, como subconjunto de C++, existen sentencias especiales para el manejo de excepciones. A continuación se muestra mediante un ejemplo sencillo la utilidad básica de dichas sentencias y los distintos pasos para el

tratamiento de exceptiones. Un subprograma desarrollado siguiendo el modelo de terminación podría programarse en **C±** según el siguiente esquema:

```
void Operacion ( argumentos ) {
  . . . .
  ... acción1 ...
  if ( error1 ) {
    throw excepcion1 /* Terminación con excepción1 */
  }
  ... acción2 ...
  if( error2 ) {
    throw excepcion2 /* Terminación con excepción2 */
  }
  . . . .
}
```

La sentencia `throw` provoca la *terminación* del subprograma de manera semejante a una sentencia `return`. Sin embargo, ambas terminaciones son distintas: con `return` se realiza una terminación normal y con `throw` se realiza una terminación por excepción. La sentencia `throw` puede devolver cualquier tipo de resultado en `excepcion`. Además, la sentencia `throw` es la encargada de indicar que se ha detectado una situación de error (actividad 1) y lanzar el mecanismo de tratamiento de excepciones. Quien utiliza el subprograma será el encargado de realizar la corrección de la situación de error (actividad 2). Aplicaremos este esquema a una variante mejorada de la función para calcular el factorial de un número, detectando la situación de exceso de capacidad (*overflow*) lanzando una excepción con el valor del número cuyo factorial provoca el *overflow*. Además, también se lanza una excepción con valor cero cuando se le pide a la función el factorial de un número negativo.

```
int FactorialRobusto(int n) {
  int f = 1;

  if (n < 0) {
    throw 0;
  }
  for (int k = 2; k <= n; k++) {
    if (f > INT_MAX/k) {
      throw k;
    }
    f = f * k;
  }
  return f;
}
```

Esta función opera de manera robusta sea cual sea el rango de enteros de la máquina. Si se sabe de antemano cuál es dicho rango, se podría aumentar algo la eficiencia determinando por anticipado cuál es el mayor valor para el que se puede calcular el factorial, y detectando directamente si el valor del argumento excede de dicho límite, definido como parámetro constante.

La segunda actividad, sin embargo, no puede realizarse, en general, dentro del subprograma, ya que el tratamiento adecuado de la situación excepcional podrá ser diferente en cada invocación. Lo que ha de hacer el subprograma es devolver una indicación precisa del error, y dejar que sean los programas que lo invocan quienes decidan cómo actuar frente al error en cada caso.

Con las herramientas convencionales de un lenguaje de programación, el esquema típico para el tratamiento de excepciones sería el siguiente:

```
Algoritmo del Problema (inicio);
OperacionRobusta( argumentos );
if (Excepcion) {
  Tratamiento de la Excepción
}
Algoritmo del Problema (continución)
```

Este esquema tiene el inconveniente de que hay que insertar el tratamiento de la excepción en medio del código del algoritmo del problema que se está resolviendo. Esta mezcla del código normal y el código excepcional disminuye la claridad del programa. Las sentencias disponibles en **C±** para el manejo de excepciones permiten separar ambos códigos siguiendo el siguiente esquema:

```
try {
  Algoritmo del Problema (inicio):
  OperaciónRobusta( argumentos );
  Algoritmo del Problema (continuación);
}
catch (Excepción) {
  Tratamiento de la Excepción
}
```

La sentencia `try` agrupa el bloque de código en el que se programa el algoritmo del problema a resolver sin tener en cuenta las posibles excepciones que se pudiran producir. A continuación, la sentencia `catch` agrupa el código para el tratamiento de la `Excepción` que se declara entre parentesis. Dentro del mismo bloque `try` se pueden producir excepciones de distintos tipos para las que se tendrian que programar las correspondientes sentencias `catch`. En el programa completo que figura a continuación se muestra como se programa el tratamiento de las excepciones que genera la función `FactorialRobusto` en el bloque que la utiliza.

```
/** Programa: FactorialRobusto */

#include <stdio.h>
#include <limits.h>

int FactorialRobusto( int n ) {
  int f = 1;

  if (n < 0) {
    throw 0;
  }
  for (int k = 2; k <= n; k++) {
    if (f > INT_MAX/k) {
      throw k;
    }
    f = f * k;
  }
  return f;
}

void EscribirFactorial( int num ) {
  try {
    printf( "%2d! vale:%10d\n", num, FactorialRobusto(num) );
  }
  catch (int e) {
    printf( "%2d! excepción: ", num );
    if (e == 0) {
      printf( "Factorial de número negativo\n" );
    } else {
      printf( "Superado límite al evaluar %2d!\n", e );
    }
  }
}

int main () {
  for (int i = 5; i >= -2; i--) {
    EscribirFactorial( i );
  }
  printf( "\n" );
  for (int i = 6; i <= 15; i++) {
    EscribirFactorial( i );
  }
}
```

El procedimiento **EscribirFactorial** captura todas excepciones y el tratamiento consiste en escribir el correspondiente mensaje de error. En este caso, el programa principal sólo se utiliza como programa de prueba. Los resultados de la ejección se muestran a continuación:

```
 5! vale:       120
 4! vale:        24
 3! vale:         6
 2! vale:         2
 1! vale:         1
 0! vale:         1
-1! excepción: Factorial de número negativo
-2! excepción: Factorial de número negativo

 6! vale:       720
 7! vale:      5040
 8! vale:     40320
 9! vale:    362880
10! vale:   3628800
11! vale:  39916800
12! vale: 479001600
13! excepción: Superado límite al evaluar 13!
14! excepción: Superado límite al evaluar 13!
15! excepción: Superado límite al evaluar 13!
```

# Tema 9

# Definición de tipos

Después de haber sido introducidos todos los mecanismos fundamentales para la construcción de programas, ahora se pasa a estudiar las estructuras de datos. En este tema se indican las primeras formas en que el programador puede definir sus propios tipos de datos.

En primer lugar se estudian los tipos escalares simples definidos por enumeración y cómo se utilizan. Como caso especial de tipo enumerado ya predefinido se hace especial mención del tipo `bool`, precisando su importancia dentro de la programación.

A continuación se estudia la *definición de tipos* estructurados y las dos formas más importantes para la estructuración de datos: *array* o formación y *struct* o registro. En este tema sólo se introducen los conceptos básicos de ambas estructuras de datos. También en este tema y como ejemplo de formación se estudian las cadenas o vectores de caracteres.

Para finalizar, se presentan varios ejemplos que emplean los tipos introducidos y muestran las posibilidades que ofrecen.

## 9.1 Tipos definidos

Una de las ventajas fundamentales de los lenguajes de alto nivel es la posibilidad que ofrecen al programador de definir sus propios tipos de datos. Los tipos predefinidos: `int`, `char` y `float`, ya presentados en el tema 2, nos han permitido la elaboración de programas para la realización de cálculos o el manejo de caracteres. Sin embargo, si se trata de realizar un programa para jugar al ajedrez resulta mucho más adecuado utilizar datos que representen de manera

más directa a los peones, caballos, torres, alfiles, reyes y damas del tablero. Razonamientos similares se pueden hacer si se quieren realizar programas que manejen días de la semana, deportes, colores, alimentos, etc.

Mediante la definición de nuevos tipos de datos por el programador se consigue que cada información que maneja el computador tenga su sentido específico. El tipo establece los posibles valores que puede tomar ese dato. Además, al igual que sucedía con los tipos predefinidos, a cada nuevo tipo que se define se asocian un conjunto de operaciones que se pueden realizar con él. Por tanto, la definición de tipos supone crear un nuevo nivel de *abstracción* dentro del programa.

En **C±** la declaración de los tipos se realiza, junto a la declaración de las constantes y variables, dentro de las *Declaraciones* del programa principal o en cualquiera de sus procedimientos o funciones. Asimismo, en **C±** la declaración de cada nuevo tipo siempre se inicia con la palabra clave `typedef`. Por ejemplo:

```
typedef int   TipoEdad;
typedef char  TipoSexo;
typedef float TipoAltura;
```

En estas declaraciones se definen nuevos tipos dándoles un nombre o identificador y haciéndolos equivalentes o *sinónimos* de otros tipos ya definidos (en este caso, los predefinidos `int`, `char` y `float`). Quizá en estos ejemplos la declaración de tipo no cubre todos los objetivos señalados anteriormente, pues no establece ninguna especificidad. Esto es, convendría establecer que la edad no puede ser negativa ni superior a un valor determinado o que el sexo sólo puede tomar determinados valores. En algunos lenguajes de programación, tales como Pascal, Modula-2 o Ada, se puede acotar el rango de valores de un tipo de datos a partir de otro en el momento de la declaración. Lamentablemente, en **C±** sólo es posible acotar el rango de valores de un dato haciendo las correspondientes comprobaciones dentro del código del programa.

Es importante señalar que igual que se han utilizado los tipos predefinidos, en la definición de un nuevo tipo se pueden utilizar (y normalmente se utilizan) otros tipos definidos previamente, según veremos a lo largo de este tema. Precisamente esta característica es la más importante de la posibilidad de declarar nuevos tipos.

La definición de tipos es solamente una declaración de los esquemas de datos que se necesitan para organizar la información de un programa. Para almacenar información es necesario declarar y utilizar variables de los correspondientes tipos, de la misma forma que se hace con los tipos predefinidos.

Por ejemplo, se podrían usar los *tipos sinónimos* anteriores de la forma:

```
TipoEdad edad1, edad2;
TipoSexo sexo;
TipoAltura altura;

edad2 = edad1 + 5;
sexo = 'V';
altura = 1.75;
```

De manera formal, la sintaxis de la declaración de tipos es la siguiente:

> *Declaración_de_tipo* **::=** *Tipo_sinónimo* **|** *Tipo_enumerado* **|**
> *Tipo_array* **|** *Tipo_struct* **|** *Tipo_unión* **|** *Tipo_puntero*

El tipo sinónimo ya ha sido utilizado en los ejemplos anteriores para introducir el concepto de tipo de dato. Formalmente la declaración de un tipo sinónimo es la siguiente:

> *Tipo_sinónimo* **::=**
> `typedef` *Identificador_de_tipo Identificador_de_tipo_nuevo* ;

El tipo sinónimo puede parecer trivial o meramente teórico, sin embargo, tiene una utilidad bastante importante como mecanismo de parametrización del programa. Al igual que sucedía con las constantes con nombre, en un programa se pueden utilizar sólo tipos con nombres propios. Por ejemplo:

```
typedef int   entero;
typedef char  caracter;
typedef float real;
```

Estos nuevos tipos `entero`, `caracter` y `real` sustituyen a los predefinidos del lenguaje y son los únicos que se utilizarán en nuestro programa. Cuando se cambia de compilador para transportar el programa a otra plataforma o se quiere cambiar la precisión de los cálculos sólo es necesario modificar en estas sentencias de parametrización los tipos haciéndolos sinónimos de otros diferentes.

En los apartados siguientes de este mismo tema se indica la manera de definir los tipos enumerados, formación y tupla. En temas posteriores se explicará cómo se definen y para qué sirven los tipos unión y puntero.

## 9.2 Tipo enumerado

Aparte de los valores predefinidos básicos (números, caracteres, etc.) en **C±** se pueden definir y utilizar nuevos valores simbólicos de la manera que se indica a continuación.

### 9.2.1 Definición de tipos enumerados

Una manera sencilla de definir un nuevo tipo de dato es enumerar todos los posibles valores que puede tomar. En **C±** el nuevo *tipo enumerado* se define detrás de la palabra clave `enum` mediante un identificador del tipo y a continuación se detalla la lista con los valores separados por comas (`,`) y encerrados entre llaves `{...}`. Cada posible valor también se describe mediante un identificador. Estos identificadores al mismo tiempo quedan declarados como valores constantes. Por ejemplo:

```
typedef enum TipoDia {
  Lunes, Martes, Miercoles, Jueves,
  Viernes, Sabado, Domingo
};
typedef enum TipoMes {
  Enero, Febrero, Marzo, Abril, Mayo,
  Junio, Julio, Agosto, Septiembre,
  Octubre, Noviembre, Diciembre
};
typedef enum TipoEstadoCivil { Casado, Soltero, Viudo, Divorciado };
typedef enum TipoColor { Rojo, Amarillo, Azul };
typedef enum TipoFrutas { Pera, Manzana, Limon, Naranja, Kiwi };
typedef enum TipoOrientacion { Norte, Sur, Este, Oeste };
typedef enum TipoPieza { Rey, Dama, Alfil, Caballo, Torre, Peon };
```

La enumeración implica un orden que se establece entre los valores enumerados. En **C±** este orden se define de forma implícita e impone que el primer elemento de la lista ocupa la posición 0, el siguiente la 1, y así sucesivamente hasta el último, que ocupa la posición $N$-1, siendo $N$ el número de elementos enumerados. Los tipos de datos enumerados forman parte de una clase de tipos de **C±** denominados *tipos ordinales*, a la cual pertenecen también los tipos `int` y `char`, pero no el tipo `float`.

La sintaxis exacta de la declaración de los tipos enumerados es la siguiente:

*Tipo_enumerado* **::=** `typedef enum` *Identificador_de_tipo_nuevo*
  `{` *Lista_de_identificadores* `};`

*Lista_de_identificadores* **::=** *Identificador* **{** , *Identificador* **}**

### 9.2.2 Uso de tipos enumerados

Los tipos enumerados se emplean de manera similar a los tipos predefinidos. El identificador de tipo se puede emplear para definir variables de ese tipo, y

los identificadores de los valores enumerados se emplean como las constantes con nombre. Usando las definiciones anteriores podremos escribir:

```
TipoDia diaSemana;
TipoColor colorCoche = Rojo;
TipoMes mes;

diaSemana = Lunes;
colorCoche = Azul;
mes = Marzo;
```

Como se puede observar, en la misma definición de la variable **colorCoche** se ha inicializado al valor **Rojo**, de forma semejante a cualquier otra variable. Puesto que entre los valores enumerados existe un orden definido, podremos emplear con ellos los operadores de comparación para programar sentencias del tipo:

```
if (mes >= Julio) {...}

while (diaSemana < Sabado) {...}

if (colorCoche = Rojo) {...}
```

Al igual que para el resto de los tipos ordinales, con los tipos enumerados se puede utilizar la notación **int(e)** para obtener la posición de un valor en la lista de valores del tipo. Por ejemplo, se cumple que:

```
int(Casado) == 0
int(Kiwi) == 4
int(Diciembre) == 11
```

La operación inversa, que permita conocer qué valor enumerado ocupa una determinada posición, se consigue mediante la notación inversa que hace uso del identificador del tipo enumerado y que se invoca de la siguiente forma:

TipoEnumerado($N$)

que devuelve el valor que ocupa la posición $N$ en la colección de valores del tipo **TipoEnumerado**. En los ejemplos anteriores se cumple que:

```
TipoEstadoCivil(0) == Casado
TipoFrutas(4) == Kiwi
TipoMes(11) == Diciembre
```

Hay tipos enumerados, tales como el **TipoColor** o el **TipoFrutas**, en que la enumeración sólo sirve para reflejar los valores posibles sin que el orden de la definición tenga ninguna relevancia. Sin embargo, como sucede en los casos

de `TipoDia` y `TipoMes`, el orden es muy importante y habitualmente se quiere utilizar en los programas para manejar los valores anteriores o posteriores. Cuando se crea un nuevo elemento en un programa es responsabilidad del programador dotarle de todas las características y atributos que se consideren necesarios. Así, para disponer de las operaciones anterior/posterior de cada nuevo tipo que se defina es necesario programarlas y para ello siempre se aprovecharán las operaciones ya disponibles. Por ejemplo, para pasar al día siguiente de la semana y al mes anterior se podría programar de la siguiente forma:

```
diaSemana = Jueves;
diaSemana = TipoDia(int(diaSemana)+1);
mes = Marzo;
mes = TipoMes(int(mes)-1);
```

El resultado será:

```
diaSemana == Viernes
mes == Febrero
```

Sin embargo estas operaciones no están completas. Si se hace esto mismo cuando el día de la semana es domingo:

```
diaSemana = Domingo;
diaSemana = TipoDia(int(diaSemana)+1);
```

se obtendrá un error, ya que no existe el día siguiente a `Domingo`. A continuación se resuelve este problema programando una función específica.

Un dato de tipo enumerado se puede pasar como argumento de procedimientos o funciones y puede ser el resultado de una función. Por ejemplo, si conocemos el día de la semana de `Hoy` y queremos calcular qué día de la semana será dentro de `N` días, podemos emplear la siguiente función:

```
TipoDia SumarDias(TipoDia Hoy, int N) {
  const int DiasSemana = 7;
  int aux;

  aux = (int(Hoy) + N) % DiasSemana;
  return TipoDia(aux);
}
```

Como se puede observar, primero se calcula el ordinal del nuevo día entre 0 y 6, según el orden establecido en la definición de `TipoDia` y finalmente se devuelve este ordinal convertido al tipo correspondiente mediante `TipoDia(aux)`. De la misma manera habría que realizar la función para restar días y para sumar y restar meses.

## 9.3 El tipo predefinido **bool**

Al introducir las estructuras de selección o iteración se han descrito sentencias de **C±** que utilizan expresiones lógicas o de condición. En ese momento se dijo, de manera informal, que el valor de una condición podía ser *cierto* o *falso*. De manera más precisa podemos indicar ahora que en **C±** existe el tipo predefinido `bool` que responde a la siguiente definición, análoga a la de un tipo enumerado:

```
typedef enum bool { false, true };
```

Esta definición no es necesario escribirla ya que está implícita en el lenguaje. El nombre `bool` es el identificador del tipo (abreviatura de *booleano*), y las constantes simbólicas `false` y `true` corresponden a los valores de verdad *falso* y *cierto*, respectivamente. Como tipo ordinal se cumple:

```
int(false) == 0
int(true) == 1
```

A partir del tipo predefinido `bool`, ahora es posible declarar variables de este tipo y utilizarlas, de forma similar al resto de variables, para guardar resultados de expresiones condicionales. Por ejemplo

```
bool bisiesto;

bisiesto = (anno % 4) == 0;  /* válido entre 1901 y 2099 */
```

Asimismo, es posible realizar operaciones entre ellas. En concreto, entre operandos booleanos (variables o no) es posible realizar las operaciones lógicas ya indicadas en el tema 5 para formar expresiones lógicas y cuyos operadores son los siguientes:

| Operación lógica | Operador C± |
|---|---|
| Conjunción (A y B) | && |
| Disyunción (A o B) | \|\| |
| Negación (no A) | ! |

Esto permite formar expresiones y sentencias tales como la siguiente:

```
if (bisiesto && (mes > Febrero)) {
  totalDias = totalDias + 1;
}
```

Los resultados de las expresiones lógicas para los distintos operandos y operadores son los siguientes:

| a | b | a && b | a \|\| b | !a |
|---|---|---|---|---|
| true | true | true | true | false |
| true | false | false | true | false |
| false | true | false | true | true |
| false | false | false | false | true |

El *tipo booleano*, como cualquier otro tipo enumerado, se puede pasar como argumento de un procedimiento o función y puede ser devuelto como resultado de una función. De hecho es frecuente definir funciones cuyo resultado es un valor booleano cuando se quiere realizar un test sobre los argumentos de la función. Este tipo de funciones se denominan *predicados*. Un ejemplo de este tipo de funciones son algunas de las introducidas en el tema 7 para el manejo de caracteres y que se recuerdan a continuación:

| | |
|---|---|
| `bool isalpha( char c )` | Indica si `c` es una letra |
| `bool isascii( char c )` | Indica si `c` es un carácter ASCII |
| `bool isblank( char c )` | Indica si `c` es un carácter de espacio o tabulación |
| `bool isdigit( char c )` | Indica si `c` es un dígito decimal (0-9) |
| `bool islower( char c )` | Indica si `c` es una letra minúscula |
| `bool isspace( char c )` | Indica si `c` es espacio en blanco o salto de línea o página |
| `bool isupper( char c )` | Indica si `c` es una letra mayúscula |

Conviene recordar que para poder usar estas funciones predicado es necesario incluir la cabecera de librería `<ctype.h>` al comienzo del programa.

## 9.4 Tipos estructurados

Todos los tipos de datos presentados hasta este momento se denominan *tipos escalares*, y son datos simples, en el sentido de que no se pueden descomponer. En general, no tiene sentido tratar de reconocer fragmentos de información independientes dentro de un valor entero, o un carácter, o el valor simbólico de un día de la semana o el número de un día del mes.

En muchas aplicaciones resulta conveniente, o incluso necesario, manejar globalmente elementos de información que agrupan colecciones de datos. Por ejemplo, puede ser apropiado manejar como un dato único el valor de una fecha que incluye la información del día, el mes y el año como elementos componentes separados. Con este objetivo, los lenguajes de programación dan la posibilidad de definir tipos de datos estructurados.

Un *tipo estructurado* de datos, o estructura de datos, es un tipo cuyos valores se construyen agrupando datos de otros tipos más sencillos. Los elementos de información que integran un valor estructurado se denominan *componentes*. Todos los tipos estructurados se definen, en último término, a partir de tipos simples combinados.

En los próximos apartados se hace una primera introducción de los tipos estructurados *formación* y *tupla*.

## 9.5 Tipo formación y su necesidad

Las estructuras de datos de tipo *formación* son quizá las más básicas, o al menos las que se introdujeron primero en los lenguajes de programación imperativos. Puede afirmarse que no hay ningún programa real interesante que no use estructuras de esta clase.

En el tema 5 se realizó un programa para ordenar 3 datos. El interés de la ordenación de cualquier tipo de dato (números, nombres, fechas, etc.) resulta más evidente cuando la cantidad de datos a ordenar es de cientos o miles de datos, el trabajo de ordenación resulta tedioso y es más adecuado realizarlo utilizando un computador.

Si se quiere realizar un programa de ordenación con las estructuras de datos presentadas hasta este momento sería necesario declarar tantas variables del mismo tipo como datos se tratan de ordenar. En el programa del tema 5 se necesitaron tres variables: `valorUno`, `valorDos` y `valorTres`. Además el tratamiento de cada variable se debe realizar por separado: dentro del texto del programa se tienen que hacer las correspondientes comparaciones e intercambios entre cada una de las parejas de variables. Por ejemplo, en el mencionado programa se tenía el siguiente fragmento:

```
/*-- Primer Paso: Ordenar 2 primeros datos --*/
if (valorUno > valorDos) {
  auxiliar = valorUno;
  valorUno = valorDos;
  valorDos = auxiliar;
}
/*-- Segundo Paso: Situar el 3º dato --*/
if (valorTres < valorUno) {
  auxiliar = valorTres;
  valorTres = valorDos;
  valorDos = valorUno;
  valorUno = auxiliar;
```

```
} else if (valorTres < valorDos) {
  auxiliar = valorDos;
  valorDos = valorTres;
  valorTres = auxiliar;
}
```

Evidentemente esta forma de realizar el programa es imposible de generalizar para la ordenación de un número cualquiera de datos. Además, este mismo problema se reproduce en cualquier programa en el que se trate de manejar una cantidad razonable de datos, todos del mismo tipo. Si tenemos en cuenta que uno de los primeros objetivos de los computadores fue precisamente manejar grandes cantidades de información, se comprende fácilmente porqué ya en los primeros lenguajes de programación se disponía de estructuras de datos para resolver estos problemas. Estas estructuras se denominan genéricamente formaciones (en inglés *array*), y permiten la generalización de la declaración, referencia y manipulación de colecciones de datos todos del mismo tipo. En el siguiente apartado se estudian los vectores como la forma más elemental de formación y sus características.

## 9.6 Tipo vector

Como se muestra en la figura 9.1, un vector está constituido por una serie de valores, todos ellos del mismo tipo, a los que se les da un nombre común que identifica a toda la estructura globalmente. Cada valor concreto dentro de la estructura se distingue por su índice o número de orden que ocupa en la serie. En la figura 9.1 se ha adoptado el convenio que se utiliza en **C±** por el que el índice del primer elemento siempre es el cero.

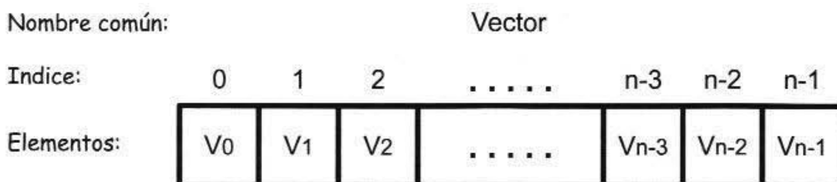



Figura 9.1 Estructura vector.

Como se puede observar, esta estructura es análoga al concepto matemático de vector, en el que el vector completo se identifica por un nombre único y cada elemento particular mediante el correspondiente subíndice:

$$V = (V_0, V_1, V_2, V_3, ...., V_{n-2}, V_{n-1})$$

En cuanto al aspecto de programación se puede establecer un paralelismo entre la estructura de programación en la que se repite la misma acción un número de veces determinado: sentencia **for**, y la estructura de datos vector en la que también se repiten un número de veces determinado el mismo tipo de dato. Como se verá posteriormente, la sentencia **for** es la que mejor se adecúa al manejo de los vectores y en general de todo tipo de formaciones.

### 9.6.1 Declaración de vectores

En **C±**, una estructura de *tipo vector* se declara de la siguiente forma:

typedef *TipoElemento* *TipoVector*[*NumeroElementos*];

donde *TipoVector* es el nombre del nuevo tipo de vector que se declara y *NumeroElementos* es un valor constante que indica el número de elementos que constituyen el vector. Por tanto, la variabilidad del índice de un vector siempre estará comprendida entre 0 y *NumeroElementos* -1. Finalmente *TipoElemento* corresponde al tipo de dato de cada uno de los elementos del vector y puede ser cualquier tipo de dato predefinido del lenguaje o definido por el programador. Indudablemente, cualquiera de los nuevos tipos de datos estructurados que se vayan presentado en este tema y los próximos, también podrán ser elementos de un vector. Los siguientes ejemplos utilizan los tipos predefinidos y algunos tipos definidos por enumeración:

```
typedef enum TipoDia {
  Lunes, Martes, Miercoles, Jueves,
  Viernes, Sabado, Domingo
};
typedef enum TipoColor { Rojo, Amarillo, Azul };
typedef float TipoMedidas[3];
typedef TipoColor TipoPaleta[5];
typedef char TipoCadena[30];
typedef TipoDia TipoAgenda[7];
typedef bool TipoEstados[8];
typedef int TipoVector[10];
```

En muchos casos el tamaño del vector es un parámetro del programa que podría tener que cambiarse al adaptarlo a nuevas necesidades. Si es así, resulta aconsejable que la declaración del número de elementos se realice como una constante con nombre. Por ejemplo, estas constantes podrían haber sido declaradas previamente de la siguiente forma:

```
const int NumeroEstados = 8;
const int LongitudAgenda = 7;
const int NumeroLetras = 30;
const int NumeroElementos = 10;

typedef char TipoCadena[NumeroLetras];
typedef TipoDia TipoAgenda[LongitudAgenda];
typedef bool TipoEstados[NumeroEstados];
typedef int TipoVector[NumeroElementos];
```

De esta manera, el programa queda parametrizado por dichas constantes. En las modificaciones posteriores, si se quiere adaptar el tamaño del vector sólo es necesario modificar esta constante. Además, como se verá posteriormente, es habitual utilizar el número de elementos del vector en las operaciones de recorrido y búsqueda de los vectores, que se pueden entonces programar en función de la misma constante.

Para poder utilizar los tipos declarados es necesario declarar a su vez, posteriormente, las correspondientes variables. Por ejemplo:

```
TipoAgenda agendaUno, agendaDos;
TipoCadena frase;
TipoEstados estadoMotor, estadoPanel;
TipoVector vectorUno, vectorDos;
```

Hay que recordar que en el lenguaje **C±** es obligatorio que todas las variables se declaren precedidas del identificador de un tipo predefinido del lenguaje o bien definido previamente en el programa. Tanto en la sintaxis de **C±** como en el *Manual de Estilo* está expresamente prohibido la declaración de variables de *tipo anónimo*. Se dice que una variable es de tipo anónimo cuando su estructura se detalla en la misma declaración de la variable, como si se estuviera declarando un tipo de datos anónimo para esta única variable.

▮ NOTA: En C/C++ sí es posible declarar variables de tipo anónimo aunque también es bastante frecuente que se aconseje o incluso se obligue en el *Manual de Estilo* que no se usen tipos anónimos para facilitar la comprensión del programa.

La sintaxis exacta de la declaración de los tipos formación es la siguiente:

*Tipo_formación* **::=** typedef *Identificador_de_tipo_elemento*
*Identificador_de_tipo_formación Dimensiones* ;

*Dimensiones* **::=** *Tamaño* **{** *Tamaño* **}**

*Tamaño* **::=** [ *Número_de_elementos* ]

En esta sintaxis está incluida la posibilidad de declarar formaciones multidimensionales o matrices que se explicarán en el tema 11.

### 9.6.2 Inicialización de un vector

Según se explicó en el tema 3, en **C±** cuando se declara una única variable de cualquier tipo siempre es posible darle un valor inicial. En el caso de un vector la inicialización afecta a todos sus elementos y por tanto la notación es algo especial y en ella se indica el valor inicial de todos los elementos agrupándolos entre llaves `{...}` y separándolos por comas (`,`). A continuación se declaran nuevamente alguna de las variables anteriores incluyendo su inicialización.

```
TipoAgenda agendaUno = {
  Lunes, Viernes, Domingo, Martes,
  Martes, Martes, Sabado
};
TipoEstados estadoMotor = {
  true, false, true, true, false,
  false, false, true
};
TipoVector vectorUno = { 12, 7, 34, -5, 0, 0, 4, 23, 9, 11 };
TipoVector miVector = { 1, 1, 1, 1, 1, 0, 0, 0, 0, 0 };
```

### 9.6.3 Operaciones con elementos de vectores

La mayoría de las operaciones interesantes con vectores hay que realizarlas operando con sus elementos uno por uno. La referencia a un elemento concreto de un vector se hace mediante el nombre del vector seguido, entre corchetes, del índice del elemento referenciado. Por ejemplo:

```
vectorUno[0]
frase[13]
estadoMotor[5]
miVector[3]
```

Un elemento de un vector puede formar parte de cualquier expresión con constantes, variables u otros elementos. Paras estas expresiones se tendrá en cuenta el tipo de los elementos del vector y las reglas de compatibilidad. Por ejemplo:

```
miVector[3] = 3*vectorUno[0] + 2*vectorDos[0];
frase[13] = 'A';
estadoMotor[5]  = true;
```

Los elementos de los vectores `miVector`, `vectorUno` y `vectorDos` son todos del mismo tipo `int` y por tanto, la primera de las expresiones anteriores es totalmente correcta.

Como índice para designar un elemento de un vector se puede utilizar una variable o expresión, siempre que sean de tipo entero. Por ejemplo:

```
const int Alarma = 0;
int i, j;
int origen, indice;

miVector[j] = 3*vectorUno[i] + 2*vectorDos[i+3];
frase[origen+indice] = ' ';
estadoPanel[Alarma] = false;
```

La posibilidad de utilizar variables o expresiones para el cálculo del índice de un vector es fundamental en cualquier programa que utilice vectores, como se verá a lo largo de este y los próximos temas. Sin embargo, siempre se debe comprobar exhaustivamente que no existe ninguna posibilidad de que por error el índice calculado por la expresión o guardado en la variable se salga fuera del rango entre 0 y *NumeroElementos*-1 para cada vector. No obstante, si esto ocurre el resultado será totalmente impredecible dado que estaremos haciendo referencia a un elemento del vector que no tiene existencia real. En esta situación, y dependiendo del compilador, se podrá producir un error de ejecución y/o la parada del programa.

Es especialmente importante insistir en que al usar lenguajes como C, C++ o **C±** la comprobación de que el índice para acceder a un elemento de vector está dentro del rango permitido es **responsabilidad del programador**. Muchos ataques informáticos aprovechan la falta de previsión de esta comprobación para alterar el funcionamiento normal de un programa suministrándole datos de mayor tamaño que el previsto y provocar lo que se denomina en inglés *buffer overrun*.

### 9.6.4 Operaciones globales con vectores

En lenguajes tales como Pascal, Modula-2, Ada, etc. existe la posibilidad de realizar una asignación global de un vector a otro, siempre que estos sean compatibles entre sí. En cualquiera de estos lenguajes, para realizar una asignación global basta escribir:

`vectorDos := vectorUno;` *{lenguaje Pascal}*

El símbolo `:=` es el operador de asignación, similar al operador = de **C±**. Esta operación efectúa una copia de todos los elementos del vector `vectorUno` en el vector `vectorDos` .

Sin embargo, en **C±** no existe esta posibilidad y la asignación se tiene que programar explícitamente mediante un bucle que realice la copia elemento a elemento, tal y como se recoge en el siguiente fragmento de programa:

```
for (int i = 0; i < NumeroElementos; i++) {
  vectorDos[i] = vectorUno[i];
}
```

Como se puede observar en el fragmento de programa anterior, se utiliza una sentencia `for` con condición de terminación `i < NumeroElementos` que no sigue la sintaxis de ninguna de las dos variantes presentadas en el tema 5. Con formaciones resulta bastante habitual realizar, como en este caso, un recorrido por todos sus elementos. Tal y como se ha dicho anteriormente, para una formación de `N` elementos, en **C±** los índices siempre van desde `0` hasta `N-1` y la forma más natural de expresar la condición de terminación del recorrido será `indice < N`. Para facilitar el trabajo con las formaciones, en **C±** se dispone de esta tercera variante de `for`. Así, la sintaxis completa de esta sentencia es la siguiente:

*Sentencia_FOR* **::=** *For_creciente* **|** *For_creciente_menor* **|**
  *For_decreciente*

*For_creciente* **::=** `for` ( `int` *Variable_índice* = *Valor_inicial* ;
  *Variable_índice* <= *Valor_final* ; *Variable_índice* ++ )
  { *Secuencia_de_sentencias* }

*For_creciente_menor* **::=** `for` ( `int` *Variable_índice* = *Valor_inicial* ;
  *Variable_índice* < *Valor_final* ; *Variable_índice* ++ )
  { *Secuencia_de_sentencias* }

*For_decreciente* **::=** `for` ( `int` *Variable_índice* = *Valor_inicial* ;
  *Variable_índice* >= *Valor_final* ; *Variable_índice* -- )
  { *Secuencia_de_sentencias* }

### 9.6.5 Paso de argumentos de tipo vector

Otra manera habitual de operar globalmente con los vectores es utilizarlos como argumentos de procedimientos o funciones. Por razones históricas y según se explicará en próximos temas, el manejo de toda clase de formaciones en C y C++ tiene ciertas peculiaridades. Debido a que nuestro lenguaje **C±** es un subconjunto de C++, también tiene algunas de esas mismas peculiaridades. Así, en **C±** el modo por defecto de paso de argumentos de tipo formación, y más concretamente de tipo vector, es el paso por referencia. Por ejemplo, si tenemos las siguientes declaraciones:

```
void LeerVector( TipoVector v ) {...}
void ConocerEstado( TipoEstados e ) {...}
```

Cuando se invocan estos procedimientos, además de emplear en la llamada argumentos reales que deben ser compatibles con el correspondiente argumento formal, hay que tener en cuenta que el argumento real puede ser modificado. Por ejemplo, con la siguiente invocación de los procedimientos anteriores:

```
LeerVector( vectorUno );
ConocerEstado( estadoMotor );
```

las variables `vectorUno` y `estadoMotor` podrán quedar modificadas después de ejecutar el correspondiente procedimiento. Es importante recordar que, según se explicó en el tema 7, para el resto de tipos de datos la opción por defecto es que el paso de argumentos es siempre por valor, los parámetros reales de la llamada pueden ser expresiones y los parámetros reales nunca se modifican. Por otro lado, también hay que recordar que en **C±**, exceptuando el caso de las formaciones, cuando un argumento formal se quiere pasar por referencia debe ir precedido del símbolo & en la cabecera de declaración del subprograma.

Por el contrario, en **C±**, cuando se utilizan argumentos de tipo formación y no se quiere que se modifiquen los parámetros reales en la llamada al procedimiento, los argumentos formales deben ir precedidos de la palabra clave `const`. Por ejemplo:

```
void EscribirVector( const TipoVector v ) {...}
void PintarEstado( const TipoEstados e ) {...}
```

Con estas declaraciones, si se invocan los procedimientos anteriores con las siguientes sentencias:

```
EscribirVector( vectorDos );
PintarEstado( estadoPanel );
```

las variables `vectorDos` y `estadoPanel` permanecerán inalteradas después de ejecutar el correspondiente procedimiento. Por tanto, cuando declaramos un argumento de tipo formación o vector precedido de la palabra clave `const` es equivalente al paso de dicho argumento por valor.

Para ser exactos, esta forma de paso de argumentos es más restrictiva que el paso por valor. En realidad el vector se pasa por referencia, pero se prohíbe usar asignaciones a sus elementos en el cuerpo del subprograma. En el paso por valor de otros tipos de datos el argumento formal se ve como variable local dentro del subprograma, y de hecho es una copia que puede modificarse

sin alterar el argumento real usado en la llamada. En cambio los argumentos de tipo vector declarados como `const` se ven como constantes dentro del subprograma, y sus elementos no pueden ser modificados en modo alguno.

▌En C y C++ la compatibilidad entre los argumentos formales y reales de tipo vector se limita a que tengan el mismo tipo de los elementos. Así, en los ejemplos anteriores los procedimientos `LeerVector` y `EscribirVector` podrían ser llamados con vectores de elementos enteros, sea cual sea la forma en que se han declarado y su tamaño. Esto introduce una clara inseguridad en el código. Por eso en **C±** se exige que el argumento real sea exactamente del mismo tipo vector que el argumento formal, aunque esta restricción se suaviza, en parte, con el empleo de vectores abiertos, tal como se verá en el tema 11. Las cadenas de caracteres que se describen en el apartado siguiente se tratan de hecho como vectores abiertos, y eso permite operar con cadenas de diferentes longitudes.

## 9.7 Vector de caracteres: Cadena (*string*)

Debido a su uso tan frecuente, en el tema 2 ya fueron presentadas las constantes de tipo cadena (en inglés, string). Sin embargo, hasta ahora no hemos podido utilizar variables de este tipo. Esto se debe a que en realidad las cadenas son vectores de caracteres y los vectores no han sido estudiados hasta este tema. Sin embargo, en todos los lenguajes es habitual que las cadenas de caracteres tengan ciertas peculiaridades que no tienen el resto de los vectores y por esta razón son objeto de este apartado específico.

En **C±** cualquier tipo vector cuya declaración sea de la forma:

```
typedef char Nombre[ N ]
```

se considera una cadena o *string*, con independencia de su longitud particular, esto es, del valor de `N`. Por tanto, el tipo definido en el apartado anterior `TipoCadena` es una cadena. La característica peculiar de las cadenas es la siguiente:

Una *cadena de caracteres* (en inglés *string*) es un vector en el que se pueden almacenar textos de diferentes longitudes (si caben). Para distinguir la longitud útil en cada momento se reserva siempre espacio para un carácter más, y se hace que toda cadena termine con un carácter nulo `'\0'` situado al final.

Por tanto, para declarar una cadena de un máximo de veinte caracteres se debe hacer de la siguiente forma:

```
typedef char Cadena20[21];
```

La declaración de variables de este tipo se hace de la forma habitual:

```
TipoCadena idioma = "inglés";
Cadena20 nombre, apellido;
TipoCadena direccion = "Gran Vía 23";
```

En este caso la inicialización se hace mediante una cadena constante semejante a las utilizadas en el tema 2 y no elemento a elemento separados por comas como para el resto de vectores. Como se puede observar, en la inicialización no es necesario asignar una cadena con los diecinueve, veinte o treinta caracteres. Los caracteres asignados ocupan posiciones seguidas desde el principio, y a continuación del último se sitúa el carácter nulo `'\0'` de final de cadena. Este carácter es especial y no se puede escribir. Lo que nunca se puede hacer es asignar más caracteres de los declarados pues se rebasarían los limites previstos y se provocaría un error.

Como la utilización de cadenas de caracteres es bastante habitual, el lenguaje C (y **C±**) dispone de la librería `string` (cabecera `<string.h>`) que facilita el manejo de las cadenas. Esta librería incluye una gran variedad de funciones y de ellas las más comunes son las siguientes:

| | |
|---|---|
| `strcpy( c1, c2 )` | Copia `c2` en `c1` |
| `strcat( c1, c2 )` | Concatena `c2` a continuación de `c1` |
| `strlen( c1 )` | Devuelve la longitud de `c1` |
| `strcmp( c1, c2 )` | Devuelve un resultado cero si `c1` y `c2` son iguales; menor que cero si `c1` precede a `c2` en orden alfabético, y mayor que cero si `c2` precede a `c1` en orden alfabético |

Al igual que con cualquier otro tipo de vector, no es posible realizar asignaciones globales entre cadenas. Por ejemplo:

```
apellido = "González";    /* ERROR en C± */
nombre = apellido;        /* ERROR en C± */
```

Para realizar esta operación usando el procedimiento general se programaría un `for` que copie elemento a elemento tal como se ha explicado en el apartado anterior. Sin embargo, esta operación se puede realizar utilizando la función de librería `strcpy` tal y como se muestra a continuación:

```
strcpy( apellido, "González" );
strcpy( nombre, apellido );
```

Al igual que las constantes, también cualquier variable de tipo cadena se puede utilizar como argumento de procedimientos o funciones. Más concretamente,

en el procedimiento de escritura **printf** se pueden utilizar constantes o variables de cadena empleando el formato de escritura %s para *string* o cadena. Por ejemplo:

```
printf("Datos: %s - %s : %s %s\n", idioma, direccion, "José", apellido);
```

produce un resultado similar al siguiente:

```
Datos: inglés - Gran Vía 23 : José González
```

El procedimiento de escritura sólo escribe el contenido de la variable hasta el primer carácter nulo '\0'.

También es posible utilizar variables de tipo cadena con el procedimiento de lectura **scanf** utilizando también el formato de lectura %s para *string* o cadena. Por ejemplo, sea el siguiente fragmento de código:

```
printf("Nombre y Apellido? ");
scanf("%s%s", &nombre, &apellido);
```

Un ejemplo de la ejecución (en pantalla) puede ser:

```
Nombre y Apellido? José González
```

Con ello se guardan en las variables **nombre** y **apellido** las cadenas "José" y "González" introducidas por teclado. Hay que tener en cuenta que con el formato %s para lectura, el carácter blanco sirve para separar las cadenas introducidas por teclado. La comprobación del valor que tienen las variables **nombre** y **apellido** es el resultado del procedimiento de escritura anterior.

Para finalizar este apartado, a continuación se muestra un programa muy sencillo que ilustra la utilización de las funciones anteriores para manejo de cadenas.

```
/** Programa: Cadenas */
/* Ejemplo de manejo de cadenas de caracteres */

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

typedef char TipoCadena[20];

int main () {
  TipoCadena nombre, apellido;
  typedef char TipoTexto[100];
  TipoTexto texto;
  int resul;
```

```
  printf( "Nombre y Apellido? " );
  scanf( "%s%s", nombre, apellido );
  printf( "Datos: %s - %s \n", nombre, apellido );
  printf( "Longitudes: %4d%4d\n", strlen(nombre), strlen(apellido) );
  strcpy( texto, nombre );
  printf( "Texto copiado: %s\n", texto );
  strcat( texto, apellido );
  printf( "Texto concatenado: %s\n", texto );
  resul = strcmp( apellido, nombre );
  if (resul == 0) {
    printf( "Nombre y Apellido iguales\n" );
  } else if (resul > 0) {
    printf( "%s es anterior a %s\n", nombre, apellido );
  }
}
```

Como se puede observar en el listado ha sido necesario incluir la cabecera de librería `<string.h>` al comienzo del programa. El resultado de una posible ejecución se muestra a continuación:

```
Nombre y Apellido? Sara Saramago
Datos: Sara - Saramago
Longitudes:    4   8
Texto copiado: Sara
Texto concatenado: SaraSaramago
Sara es anterior a Saramago
```

## 9.8 Tipo tupla y su necesidad

Otra forma de construir un dato estructurado consiste en agrupar elementos de información usando el esquema de *tupla* o *agregado*. En este esquema el dato estructurado está formado por una colección de componentes, cada uno de los cuales puede ser de un tipo diferente.

Por ejemplo, una fecha se describe habitualmente como un dato compuesto de los elementos día, mes y año. Un punto en el plano cartesiano se describe mediante dos números, que son sus coordenadas. El nombre completo de una persona es la colección formada por el nombre de pila y sus dos apellidos. Como ejemplos concretos podemos poner:

| Dato | Valor |
|---|---|
| fecha | < 12, Octubre, 1992 > |
| punto | < 4, -2 > |
| nombre_completo | < Fernando, Jiménez, Rodríguez > |

En estos ejemplo se ha supuesto un orden implícito entre las componentes. En la fecha se ha escrito primero el día, luego el mes, y luego el año. Es importante identificar claramente a qué componente corresponde cada elemento de información. El punto <4, -2> es distinto del punto <-2, 4>.

En realidad el orden es hasta cierto punto arbitrario. Normalmente cada componente se identifica mediante un nombre simbólico. En los ejemplos anteriores podríamos nombrar las componentes de la forma:

| Dato | Valor |
|---|---|
| fecha | < día: 12, mes: Octubre, año: 1992 > |
| punto | < x: 4, y: -2 > |
| nombre_completo | < nombre: Fernando, apellido1: Jiménez, apellido2: Rodríguez > |

Identificando cada componente por su nombre, se pueden escribir en el orden que convenga:

| Dato | Valor |
|---|---|
| fecha | < mes: Octubre, día: 12, año: 1992 > |
| punto | < x: 4, y: -2 > |
| nombre_completo | < apellido1: Jiménez, apellido2: Rodríguez, nombre: Fernando > |

Tras las consideraciones anteriores, podríamos dar una definición de este esquema de datos:

*Tupla*: Colección de elementos componentes, de diferentes tipos, cada uno de los cuales se identifica por un nombre.

Un aspecto importante del empleo de datos estructurados corresponde al punto de vista de *abstracción*. Una tupla, como cualquier otro dato compuesto, puede verse de forma abstracta como un todo, prescindiendo del detalle de sus componentes. La posibilidad de hacer referencia a toda la colección de elementos mediante un nombre único correspondiente al dato compuesto simplifica en muchos casos la escritura del programa que lo maneja.

## 9.9 Tipo registro (*struct*)

Los esquemas de tupla pueden usarse en programas en **C±** definiéndolos como estructuras del tipo *registro* o `struct`. Un registro o `struct` es una estructura

de datos formada por una colección de elementos de información llamados *campos*.

### 9.9.1 Definición de registros

La declaración de un tipo registro en **C±** se hace utilizando la palabra clave `struct` de la siguiente la forma:

```
typedef struct Tipo-registro {
  Tipo-campo-1 nombre-campo-1;
  Tipo-campo-2 nombre-campo-2;
  ...
  Tipo-campo-N nombre-campo-N;
};
```

Cada una de las parejas *Tipo-campo* y *nombre-campo*, separadas por punto y coma (`;`), define un campo o elemento componente y su correspondiente tipo. Además hay que tener en cuenta que la estructura acaba siempre con punto y coma (`;`) Como ejemplos de definiciones tenemos:

```
typedef enum TipoMes {
  Enero, Febrero, Marzo, Abril, Mayo,
  Junio, Julio, Agosto, Septiembre,
  Octubre, Noviembre, Diciembre
};
typedef struct TipoFecha {
  int dia;
  TipoMes mes;
  int anno;
};
typedef struct TipoPunto {
  float x;
  float y;
};
```

La sintaxis de la declaración de los tipos registro es la siguiente:

*Tipo_registro* **::=** `typedef struct` *Identificador_de_tipo_nuevo*
`{` *Lista_de_campos* `};`

*Lista_de_campos* **::=** *Campo* `;` **{** *Campo* `;` **}**

*Campo* **::=** *Identificador_de_tipo Identificador_de_campo*

### 9.9.2 Variables de tipo registro y su inicialización

Como siempre, para declarar variables de tipo registro es necesario haber realizado previamente la definición del tipo del registro. Hay que recordar que el *Manual de Estilo* de **C±** no permite declarar variables de *tipo anónimo*. A continuación se detalla una declaración de variables:

```
TipoFecha ayer, hoy;
TipoPunto punto1, punto2;
```

También estas variables se pueden inicializar en la declaración de una manera semejante a las formaciones agrupando los valores iniciales entre llaves `{...}` y separándolos por una coma (`,`), pero teniendo en cuenta el tipo de dato de cada campo. A continuación se declaran nuevamente alguna de las variables anteriores incluyendo su inicialización.

```
TipoFecha hoy = { 12, Marzo, 2009 };
TipoPunto punto1 = { 12.5, -76.9 };
```

### 9.9.3 Uso de registros

Al manejar datos estructurados de tipo registro se dispone de dos posibilidades: operar con el dato completo, o bien operar con cada campo por separado. Las posibilidades de operar con el dato completo son bastante limitadas. La única operación admisible es la de asignación. El valor de un dato de tipo registro puede asignarse directamente a una variable de su mismo tipo. Por ejemplo, con las definiciones anteriores es posible escribir:

```
punto2 = punto1;
```

En estas asignaciones debe cumplirse la compatibilidad de tipos. Dos estructuras son compatibles en asignación si son del mismo tipo, o de tipos sinónimos. No es suficiente la llamada compatibilidad estructural; es decir, dos estructuras con los mismos campos no son compatibles si sus definiciones se hacen por separado.

También es posible pasar como argumento un dato de tipo registro a una función o procedimiento. Por ejemplo, podemos especificar los subprogramas siguientes:

```
/* Leer el día, mes y año */
void LeerFecha( TipoFecha & fecha ) {...}

/* Distancia entre p1 y p2 */
float Distancia( TipoPunto p1, TipoPunto p2 ) {...}
```

Las operaciones de tratamiento de estructuras registro consisten normalmente en operar con sus campos por separado. La forma de hacer referencia a un campo es mediante la notación:

*registro*.*campo*

Cada campo se puede usar como cualquier otro dato del correspondiente tipo; es decir, se pueden usar los valores de los campos en expresiones aritméticas, se puede asignar valor a cada uno de los campos de una variable de tipo registro, y se pueden pasar los campos como argumentos en llamadas a subprogramas. Como ejemplo daremos una posible definición de los subprogramas anteriores:

```
/* Leer el día, mes y año */
void LeerFecha( TipoFecha & fecha ) {
  int aux;

  scanf( "%d", &aux );
  if (( aux > 0 ) && ( aux <= 31 )) {
    fecha.dia = aux;
  } else {
    fecha.dia = 1;
  }
  LeerMes( fecha.mes );
  scanf( "%d", &aux );
  fecha.anno = aux;
}

/* Distancia entre p1 y p2 */
float Distancia( TipoPunto p1, TipoPunto p2 ) {
  float dx, dy;

  dx = p2.x - p1.x;
  dy = p2.y - p1.y;
  return sqrt( dx*dx + dy*dy );
}
```

También es posible devolver como resultado un valor estructurado de tipo registro. Por ejemplo, es mucho más claro y natural escribir:

```
TipoPunto PuntoMedio( TipoPunto p1, TipoPunto p2 ) {
  TipoPunto m;

  m.x = (p1.x + p2.x) / 2.0;
  m.y = (p1.y + p2.y) / 2.0;
  return m;
}

TipoPunto a, b, centro;

centro = PuntoMedio( a, b );
```

## 9.10 Ejemplos de programas

Para finalizar este tema se recogen en este apartado varios ejemplos que utilizan los nuevos tipos introducidos.

### 9.10.1 Ejemplo: Cálculo del día de la semana de una fecha

Con este programa se calcula qué día de la semana corresponde a una fecha cualquiera que se introduce como dato. Existen algoritmos que mediante un cálculo matemático inmediato, permiten obtener el día de la semana para cualquier fecha del calendario (ver *algoritmo de congruencia de Zeller*). Sin embargo, este ejemplo es un ejercicio de programación y se utilizarán directamente las reglas con las que se conforma cada calendario. Para el cálculo se supone conocido que el día 31 de Diciembre de 1988 fue Sábado. El programa sirve para fechas desde el año 1989 hasta el año 2088.

Las declaraciones de los días de la semana y los meses son las empleadas en los apartados anteriores. Asimismo, se utiliza la función `SumarDias` descrita en el apartado 9.2.2 para calcular el día de la semana que será dentro de N días conociendo el día de hoy.

La función `DiadelaSemana` realiza el cálculo del día de la semana teniendo en cuenta los siguientes aspectos:

- El desfase en días de la semana que se introduce para cada mes, respecto al mismo día del mes de Enero. Por ejemplo, el mes de Febrero son 3, el mes de Marzo 3, .., el mes de Julio 6, etc.
- Los años bisiestos son los múltiplos de 4.
- Si el año es inferior a 89 se considera que es posterior al 2000.
- Cada año bisiesto pasado incrementa en 1 día el desfase.

El procedimiento **EscribirDia** escribe el tipo de día resultante. El listado del programa completo es el siguiente:

```
/****************************************************
* Programa: Calendario
*
* Descripción:
*   Programa para el cálculo del día de la semana
*   que corresponde a una fecha comprendida entre:
*   1/1/1989 y 31/12/2088
****************************************************/
#include <stdio.h>

typedef enum TipoDia {
  Lunes, Martes, Miercoles, Jueves,
  Viernes, Sabado, Domingo
};
typedef enum TipoMes {
  Enero, Febrero, Marzo, Abril, Mayo,
  Junio, Julio, Agosto, Septiembre,
  Octubre, Noviembre, Diciembre
};
typedef struct TipoFecha {
  int dia;
  TipoMes mes;
  int anno;
};

/*==================================================
  Función para sumar días de la semana cíclicamente
  ==================================================*/
TipoDia SumarDias( TipoDia dia, int n ) {
  const int DiasSemana = 7;
  int aux;

  aux = (int(dia) + n) % DiasSemana;
  return TipoDia(aux);
}

/*========================================
  Función para calcular el día de la semana
  que corresponde a una fecha
  ========================================*/
TipoDia DiaDeLaSemana( TipoFecha fecha ) {
  const int OrigenA = 89;
  TipoDia TreintaUnoDiciembre88 = Sabado;
```

```
bool bisiesto;
int IncreBisis, IncreAnnos, IncreDias;
TipoMes M = fecha.mes;
int A = fecha.anno;

if (M == Enero) {
  IncreDias = 0;
} else if (M == Febrero) {
  IncreDias = 3;
} else if (M == Marzo) {
  IncreDias = 3;
} else if (M == Abril) {
  IncreDias = 6;
} else if (M == Mayo) {
  IncreDias = 1;
} else if (M == Junio) {
  IncreDias = 4;
} else if (M == Julio) {
  IncreDias = 6;
} else if (M == Agosto) {
  IncreDias = 2;
} else if (M == Septiembre) {
  IncreDias = 5;
} else if (M == Octubre) {
  IncreDias = 0;
} else if (M == Noviembre) {
  IncreDias = 3;
} else {
  IncreDias = 5;
}
bisiesto = (A % 4) == 0;
if (A < OrigenA) {
  A = A + 100;              /* Año posterior al 2000 */
}
IncreAnnos = A - OrigenA;  /* Años pasados desde el 89 */
IncreBisis = IncreAnnos/4; /* Bisiestos pasados */
IncreDias = IncreDias + fecha.dia + IncreAnnos + IncreBisis;
if (bisiesto && (M > Febrero)) {
  IncreDias++;
}
return SumarDias( TreintaUnoDiciembre88, IncreDias );
}
```

```
/*==============================================
  Procedimiento para escribir el día de la semana
  ==============================================*/
void EscribirDia( TipoDia S ) {

  if (S == Lunes) {
    printf( "Lunes" );
  } else if (S == Martes) {
    printf( "Martes" );
  } else if (S == Miercoles) {
    printf( "Miércoles" );
  } else if (S == Jueves) {
    printf( "Jueves" );
  } else if (S == Viernes) {
    printf( "Viernes" );
  } else if (S == Sabado) {
    printf( "Sábado" );
  } else {
    printf( "Domingo" );
  }
}

/*==============================================
  Procedimiento para leer una fecha (mes en número)
  ==============================================*/
void LeerFecha( TipoFecha & fecha ) {
  int mes;

  scanf( "%d/%d/%d", &fecha.dia, &mes, &fecha.anno );
  fecha.mes = TipoMes( mes-1 );
  if (fecha.anno >= 100) {
    fecha.anno = fecha.anno % 100;
  }
}

/*====================================
  Procedimiento para escribir una fecha
  ====================================*/
void EscribirFecha( TipoFecha fecha ) {
  printf( "%02d/%02d/%02d", fecha.dia, int(fecha.mes+1), fecha.anno );
}
```

```
/*=================
  Programa principal
  =================*/
int main() {
  TipoFecha fecha;       /* fecha a manejar */
  char tecla;            /* tecla pulsada */
  TipoDia Hoy;
  TipoMes Pasado;

  Hoy = Miercoles;
  Pasado = Noviembre;
  tecla = 'S';
  while (tecla != 'N') {
    printf( "¿Dia Mes Año(DD/MM/AA)? " );
    LeerFecha( fecha );
    printf( "Fecha: " );
    EscribirFecha( fecha );
    printf( "   Día de la semana: " );
    EscribirDia( DiaDeLaSemana( fecha ));
    tecla = ' ';
    printf( "\n\n¿Otra Fecha(S/N)? " );
    while ((tecla != 'S') && (tecla != 'N')) {
      scanf( "%c", &tecla );
    }
  }
}
```

Un resultado de la ejecución del programa es el siguiente:

```
¿Dia Mes Año(DD/MM/AA)? 1/4/1993
Fecha: 01/04/93   Día de la semana: Jueves

¿Otra Fecha(S/N)? S
¿Dia Mes Año(DD/MM/AA)? 25/7/97
Fecha: 25/07/97   Día de la semana: Viernes

¿Otra Fecha(S/N)? S
¿Dia Mes Año(DD/MM/AA)? 29/2/2000
Fecha: 29/02/00   Día de la semana: Martes

¿Otra Fecha(S/N)? S
¿Dia Mes Año(DD/MM/AA)? 1/3/0
Fecha: 01/03/00   Día de la semana: Miércoles
```

```
¿Otra Fecha(S/N)? S
¿Dia Mes Año(DD/MM/AA)? 13/3/2009
Fecha: 13/03/09   Día de la semana: Viernes

¿Otra Fecha(S/N)? N
```

### 9.10.2 Frases palíndromas

Este programa ilustra el uso de vectores de caracteres. Se trata de comprobar si una frase es un palíndromo, esto es, si las letras de las que consta se leen igual hacia adelante que hacia atrás. Para ello se define un procedimiento `LeerTexto`, capaz de leer un vector de hasta 100 caracteres. Este procedimiento sólo guarda las letras válidas de la 'a' a la 'z' incluyendo la 'ñ' (mayúsculas o minúsculas) y finaliza cuando se introduce un punto '.' o se llena el vector pasado como argumento. Este procedimiento utiliza `scanf` leyendo carácter a carácter dando el tratamiento específico que se desea, en el que sólo se guardan los caracteres mencionados.

La función predicado `Simetrico` es la encargada de comprobar que el contenido del vector, pasado como argumento por valor, es simétrico. Primeramente, se busca el punto de final del texto a analizar o el final del vector. La posición final se decrementa en uno después de la búsqueda para apuntar al primer carácter válido. Después, se comprueba si coinciden los caracteres de un extremo con los del otro. Esta comprobación acaba o bien cuando se cruzan los índices y todos los caracteres han sido iguales o bien cuando no coinciden algunos de ellos.

El listado completo del programa es el siguiente:

```
/*******************************************
* Programa: Palindromo
*
* Descripción:
*   Este programa comprueba si una frase es
*   un palíndromo
********************************************/
#include <stdio.h>
#include <ctype.h>

const char Fin = '.';   /* carácter final de la frase */
const int Maximo = 100; /* máxima longitud de la frase */
typedef char TipoCadena[Maximo];
```

```
/*=========================================
  Procedimiento para leer una frase acabada
  en punto (.) y dejarla en el argumento.
  Sólo se guardan en el vector argumento las
  letras a..z mayúsculas o minúsculas
  incluyendo la ñ y Ñ y el punto final.
  =========================================*/
void LeerTexto( TipoCadena texto ) {
  int longitud = 0;
  char caracter = ' ';

  /*-- Leer y guardar sólo las letras de la frase --*/
  printf( "¿Frase acabada en punto(.)?\n" );
  while ((caracter != Fin) && (longitud < Maximo)) {
    scanf("%c",&caracter);
    if (((caracter>='a') && (caracter<='z')) ||
        ((caracter>='A') && (caracter<='Z')) ||
        (caracter==Fin) || (caracter=='ñ') ||
        (caracter=='Ñ')) {
      texto[longitud] = toupper(caracter);
      longitud++;
    }
  }
}

/*===============================================
  Función que comprueba si coinciden las letras en
  posiciones simétricas: La frase es un palíndromo
  ===============================================*/
bool Simetrico( const TipoCadena Texto ) {
  int i, j;

  /*-- 1º: Buscar el punto final --*/
  j = 0;
  while ((Texto[j] != Fin) && (j<Maximo)) {
    j++;
  }
  j--;

  /*-- 2º: Comprobar igualdad entre simétricos --*/
  i = 0;
  while ((i < j) && (Texto[i] == Texto[j])) {
    i++;
    j--;
  }
```

```
  /*-- 3º: Es palíndromo si coincide todo --*/
  return i >= j;
}

/*=================
  Programa principal
  =================*/
int main() {
  TipoCadena frase;
  LeerTexto( frase );
  if (Simetrico( frase )) {
    printf(" Es Palíndromo\n");
  } else {
    printf(" No es Palíndromo\n");
  }
}
```

El resultado de la ejecución del programa es el siguiente:

```
¿Frase acabada en punto(.)?
Dabale arroz a la zorra el abad.
 Es Palíndromo
```

### 9.10.3 Ejemplo: Cálculos con fracciones

Como ejemplo de empleo de registros, se reescribe completo el programa del tema 8 para realizar cálculos con fracciones. La nueva versión utiliza siempre funciones para devolver un resultado de tipo registro. Esta solución es más natural y nos introduce el concepto de nuevo de tipo de dato (`TipoFraccion`) y sus operaciones. El listado completo del programa se muestra a continuación:

```
/**************************************************
* Programa: Fracciones2
*
* Descripción:
*   Este programa es una calculadora que suma,
*   resta, multiplica y divide fracciones.
*   Solución utilizando registros y funciones.
**************************************************/
#include <stdio.h>

typedef struct TipoFraccion {
  int numerador;
  int denominador;
};
```

```
/*============================================
  Procedimiento para simplificar una fracción.
  Devuelve la fracción f reducida
  ============================================*/
void ReducirFraccion( TipoFraccion & f ) {
  int divisor = 2;

  while ((divisor <= f.numerador) &&
        (divisor <= f.denominador)) {
    while ((f.numerador % divisor == 0) &&
          (f.denominador % divisor == 0)) {
      f.numerador = f.numerador / divisor;
      f.denominador = f.denominador / divisor;
    }
    divisor++;
  }
}

/*====================================
  Función para sumar fracciones.
  Devuelve reducida la suma de f1 y f2
  ====================================*/
TipoFraccion SumarFracciones( TipoFraccion f1, TipoFraccion f2 ) {
  TipoFraccion suma;

  suma.numerador = f1.numerador*f2.denominador +
                   f2.numerador*f1.denominador;
  suma.denominador = f1.denominador*f2.denominador;
  ReducirFraccion( suma );
  return suma;
}

/*=====================================
  Función para restar fracciones.
  Devuelve reducida la resta de f1 y f2
  =====================================*/
TipoFraccion RestarFracciones( TipoFraccion f1, TipoFraccion f2 ) {

  f2.numerador = - f2.numerador;
  return SumarFracciones( f1, f2 );
}
```

```
/*====================================
  Función para multiplicar fracciones.
  Devuelve reducido el producto de f1 y f2
  ====================================*/
TipoFraccion MultiplicarFracciones( TipoFraccion f1, TipoFraccion f2 ) {
  TipoFraccion producto;

  producto.numerador = f1.numerador * f2.numerador;
  producto.denominador = f1.denominador * f2.denominador;
  ReducirFraccion( producto );
  return producto;
}

/*==========================================
  Función para dividir fracciones.
  Devuelve reducido el cociente de f1 y f2
  ==========================================*/
TipoFraccion DividirFracciones( TipoFraccion f1, TipoFraccion f2 ) {
  TipoFraccion cociente;

  cociente.numerador = f1.numerador*f2.denominador;
  cociente.denominador = f1.denominador*f2.numerador;
  ReducirFraccion( cociente );
  return cociente;
}

/*====================================
  Procedimiento que lee una fracción.
  Devuelve reducida la fracción leída
  ====================================*/
void LeerFraccion( TipoFraccion & f ) {
  scanf( "%d/%d", &f.numerador, &f.denominador );
  ReducirFraccion( f );
}

/*========================
  Procedimiento que escribe
    una fracción f
  ========================*/
void EscribirFraccion( TipoFraccion f ) {
  printf( "%d/%d\n", f.numerador, f.denominador );
}
```

```
/*=========================
  Programa principal
=========================*/
int main() {
  TipoFraccion acumulador = { 0, 0 }; /* acumulador = 0/0 */
  TipoFraccion operando;               /* nuevo operando a utilizar */
  char operacion = ' ';                /* tecla de operación */

  while (operacion != 'F') {
    printf( ">> " );
    scanf( " %c", &operacion );

    if (operacion == '+') {
       LeerFraccion( operando );
       acumulador = SumarFracciones( acumulador, operando );

    } else if (operacion == '-') {
       LeerFraccion( operando );
       acumulador = RestarFracciones( acumulador, operando );

    } else if (operacion == '*') {
       LeerFraccion( operando );
       acumulador = MultiplicarFracciones( acumulador, operando );

    } else if (operacion == '/') {
       LeerFraccion( operando );
       acumulador = DividirFracciones( acumulador, operando );

    } else if (operacion == 'N') { /* Nuevos cálculos */
       LeerFraccion( acumulador );

    } else if (operacion == '=') {
       printf("          ");
       EscribirFraccion( acumulador );

    } else if (operacion != 'F') {
       printf( "Pulse +, -, *, /, N, =, o F\n" );
    }
  }
}
```

En este ejemplo cada fracción se almacena como un registro con dos campos, correspondientes al numerador y al denominador, respectivamente. Comparando esta versión con la del tema 8 se aprecia que las cabeceras de los subprogramas, y por tanto las llamadas, son ahora más sencillas y naturales. El

precio a pagar es una ligera complicación al tener que hacer referencia a los valores de numerador y denominador como campos de registros, y no directamente como argumentos separados.

El resultado de una ejecución de ejemplo se muestra a continuación:

```
>> j
Pulse +, -, *, /, N, =, o F
>> N 45/78
>> =
        15/26
>> +6/9
>> =
        97/78
>> -4/5
>> =
        173/390
>> *2/3
>> =
        173/585
>> F
```

# Tema 10

# Ampliación de estructuras de control

Antes de pasar al estudio de las estructuras de datos complejas, se completa el repertorio de estructuras de control más frecuentes en los lenguajes imperativos, detallando las construcciones adoptadas en **C±**.

Se presentan algunas variantes para realizar la iteración y la selección, derivadas de las estructuras fundamentales, mostrando la posibilidad que existe de programarlas en función de ellas.

## 10.1 Estructuras complementarias de iteración

Como se explicó en el tema 5, la *programación estructurada* propone la utilización del WHILE como estructura iterativa fundamental. Esta estructura basta para poder realizar cualquier programa. Sin embargo, en el mismo tema 5 también fue introducida la estructura FOR cuya utilidad fundamental es programar un número de iteraciones que se conoce a priori y que no depende de la evolución de los cálculos realizados en cada iteración. La razón de que exista el FOR prácticamente en todos los lenguajes, es precisamente lo frecuente de este tipo de iteraciones y la facilidad que proporciona su uso en estos casos.

Las estructuras iterativas que se explican en este apartado están también disponibles habitualmente en todos los lenguajes y pretenden facilitar la programación de situaciones concretas y frecuentes. Utilizaremos en las explicaciones las sentencias disponibles en **C±** para programar las nuevas estructuras.

Será objeto de un apartado especial la sentencia **continue** de **C±** que se puede utilizar con cualquier tipo de bucle. Esta sentencia salta a la siguiente iteración, lo que es necesario cuando no tiene sentido completar todos los cálculos previstos en la iteración en curso.

### 10.1.1 Repetición: Sentencia DO

A veces resulta más natural comprobar la condición que controla las iteraciones al finalizar cada una de ellas, en lugar de hacerlo al comienzo de las mismas. En este caso, como se muestra en la figura 10.1, siempre se ejecuta al menos una primera iteración.

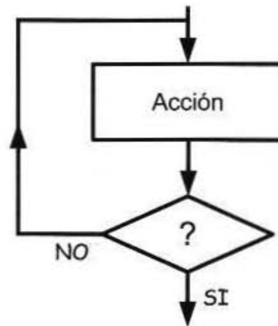



Figura 10.1 Repetición.

El formato de la estructura de repetición en **C±** es el siguiente:

```
do {
  Acción
} while ( Condición );
```

La *Condición* que controla las repeticiones es una expresión cuyo resultado es un valor de tipo `bool`. Si el resultado es `true` se vuelve a ejecutar la *Acción* y cuando el resultado es `false` finaliza la ejecución de la estructura.

Una situación típica en que resulta cómodo el empleo de esta sentencia es la que se produce cuando al finalizar cada iteración se pregunta al operador si desea continuar con una nueva. En todos estos casos, el programa siempre ejecuta la primera iteración y pregunta si se desea o no realizar otra más. Por ejemplo:

```
do {
  ...
  Operación
  ...
  printf( "¿Otra operación (S/N)?" );
  scanf( " %c", &tecla );
} while ( tecla == 'S');
```

En todos los programas realizados en los temas anteriores se ha podido programar esta forma de operar utilizando la sentencia `while`. Sin embargo, en los programas desarrollados ha sido necesario forzar la primera iteración inicializando la variable que controlaba la iteración a un valor igual al necesario para la iteración. Para el mismo ejemplo anterior, esto se realizaría de la siguiente forma:

```
tecla = 'S';
while ( tecla == 'S') {
  ...
  Operación
  ...
  printf( "¿Otra operación (S/N)?" );
  scanf( " %c", &tecla );
};
```

Esta solución es menos elegante y además tiene como inconveniente la necesidad de inicializar la variable de control, con lo que en caso de olvido, la ejecución puede ser impredecible. Como se sabe, las variables pueden tomar aleatoriamente cualquier valor inicial y, dependiendo del mismo, se ejecutará o no la primera iteración. Para estas situaciones es muy aconsejable la utilización de una estructura de repetición que evita todos estos problemas.

También resulta adecuado el empleo de la repetición cuando solamente son válidos unos valores concretos para una determinada respuesta. Si la respuesta no es correcta se solicitará de nuevo y no se continuará hasta obtener una respuesta dentro de los valores válidos. La filosofía en este caso es prácticamente la misma del caso anterior, por ejemplo:

```
do {
  printf( "¿Mes Actual?" );
  scanf( "%d", &mes );
} while ((mes < 1) || (mes > 12));
```

Evidentemente la utilidad de la repetición no está ligada exclusivamente a los casos indicados. En general, es aconsejable su uso cuando se sepa que al menos es necesaria una iteración y por tanto utilizando la sentencia `while` es necesario forzar las condiciones para que dicha iteración se produzca.

### 10.1.2 Sentencia CONTINUE

La sentencia `continue` dentro de cualquier clase de bucle (`while`, `for` o `do`) finaliza la iteración en curso e inicia la siguiente iteración. A veces, dependiendo de la evolución de los cálculos realizados en una iteración, no tiene sentido completar la iteración que se está realizando y resulta más adecuado iniciar una nueva. Esto puede suceder cuando alguno de los datos suministrados para realizar un cálculo es erróneo y puede dar lugar a una operación imposible (división por cero, raíz de un número negativo, etc.). En este caso lo adecuado es detectar la situación y dar por finalizada la iteración e iniciar una nueva iteración con nuevos datos de partida. Por ejemplo, supongamos que tenemos un vector de N coeficientes por los que hay que dividir al realizar un cierto cálculo salvo obviamente cuando uno de los coeficientes sea cero. Esto se programaría de la siguiente manera:

```
for (int i = 0; i < N; i++) {
  ...
  if ( vectorCoeficientes[i] == 0) {
    continue;
  }
  ...
  calculo = calculo / vectorCoeficientes[i];

}
```

Como se puede ver en el ejemplo, la sentencia `continue` siempre estará incluida dentro de otra sentencia condicional puesto que en caso contrario nunca se ejecutaría la parte de la iteración posterior a la sentencia `continue`. En un `for`, tal como el del ejemplo anterior, al finalizar la iteración se comprueba la condición de terminación `i < N` y si no es cierta se incrementa el índice `i++` antes de iniciar la siguiente iteración. En los otros bucles sólo se comprueba la condición de finalización para ver si procede seguir con una nueva iteración.

## 10.2 Estructuras complementarias de selección

Para la selección entre varias alternativas es suficiente disponer de la estructura IF, estudiada en el tema 5. De hecho, existen lenguajes en los que la única sentencia disponible para la selección es la propuesta por la *programación estructurada*, que permite solamente la selección entre dos alternativas. La falta de claridad cuando se utilizan varias selecciones anidadas aconseja disponer de una sentencia de selección en cascada como la estudiada en el tema 5.

Por las mismas razones de claridad y sencillez es habitual disponer de una sentencia que permite una selección por casos. Este apartado está dedicado exclusivamente a dicha sentencia, estudiando la sintaxis y semántica que tiene en **C±**.

### 10.2.1 Sentencia SWITCH

Cuando la selección entre varios casos alternativos depende del valor que toma una determinada variable o del resultado final de una expresión, es necesario realizar comparaciones de esa misma variable o expresión con todos los valores que puede tomar, uno por uno, para decidir el camino a elegir. Así, en el programa para el cálculo del día de la semana del tema anterior con la variable M que guarda el mes teníamos:

```
if (M == Enero) {
  IncreDias = 0;
} else if (M == Febrero) {
  IncreDias = 3;
} else if (M == Marzo) {
  IncreDias = 3;
} else if (M == Abril) {
  IncreDias = 6;
...

} else if (M == Octubre) {
  IncreDias = 0;
} else if (M == Noviembre) {
  IncreDias = 3;
} else {
  IncreDias = 5;
}
```

Esto supone una sentencia larga y reiterativa, si se tiene en cuenta que además para algunos casos, por ejemplo los meses de Febrero, Marzo y Noviembre, se repite la misma acción.

Si lo que se necesita es comparar el resultado de una expresión, dicha expresión se reevaluará tantas veces como comparaciones se deben realizar (a menos que se disponga de un compilador optimizante). En este caso y por razones de simplicidad y eficiencia es aconsejable guardar el resultado de la expresión en una variable auxiliar y realizar las comparaciones con dicha variable utilizando el esquema de selección por casos mostrado en la figura 10.2.

Si el tipo de valor que determina la selección es un tipo ordinal: `int`, `char` o enumerado, se dispone en **C±** de la sentencia `switch` cuya estructura permite

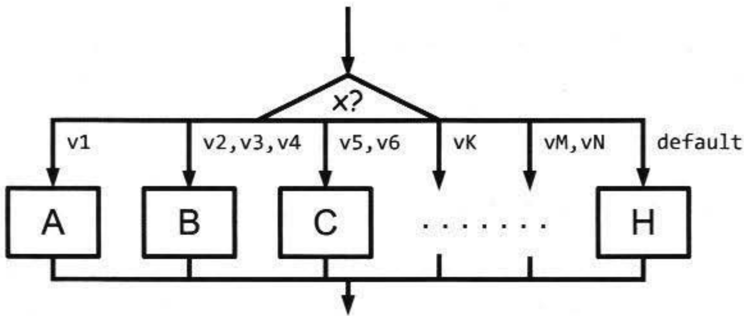



Figura 10.2. Selección por casos.

agrupar los casos que tienen el mismo tratamiento y en la que se evalúa solamente una vez la expresión `x`. La sintaxis se ilustra con el siguiente ejemplo, que muestra cómo las distintas vías de ejecución se asocian a grupos de valores que pueda tomar la expresión o variable `x`: *acciónA* con el valor $v_1$, *acciónB* con los valores de $v_2$, $v_3$ y $v_4$, *acciónC* con los valores $v_5$ y $v_6$, y estableciendo como vía alternativa la *acciónH* para el resto de valores distintos de los anteriores:

```
switch (expresión) {
  case valor1:
    acciónA;
    break;
  case valor2:
  case valor3:
  case valor4:
    acciónB;
    break;
  case valor5:
  case valor6:
    acciónC;
    break;
  . . . .
  default:
    acciónH;
}
```

La sentencia comienza con la palabra clave `switch` y a continuación, entre paréntesis, se indica la expresión o variable cuyo valor determina los casos que se quieren analizar, seguida del símbolo de abrir llave (`{`). Para cada vía de ejecución posible se detallan primeramente los valores que debe tomar la expresión, precedidos por la palabra clave `case` y seguido por dos puntos

(:). La correspondiente acción como una secuencia de sentencias se detalla a continuación de los correspondientes valores.

El *Manual de Estilo* para **C±** impone que cada acción finaliza siempre con la sentencia **break** para que finalice la sentencia `switch` después de cada acción. Si por error se omite la sentencia **break**, se continuaría ejecutando la acción del siguiente o siguientes casos. Aunque en C/C++ el uso del **break** es opcional, en **C±** es obligatorio para lograr una *programación estructurada* fácil de entender y verificar. Además, se evita la ambigüedad de si la ausencia del **break** es intencionada o es una omisión por error.

La alternativa para el resto de los valores es opcional, y va precedida de la palabra clave **default**. La sentencia finaliza con el símbolo de cerrar llave (}).

Esta sentencia no se puede utilizar cuando la variable o el resultado de la expresión que controla la selección sea de tipo `float` u otro tipo no simple. En estos casos no queda más remedio que emplear la sentencia de selección general o en cascada.

Como ejemplo se reescribe la selección anterior, según el mes:

```
switch (M) {
  case Enero:
  case Octubre:
    IncreDias = 0;
    break;
  case Mayo:
    IncreDias = 1;
    break;
  case Agosto:
    IncreDias = 2;
    break;
  case Febrero:
  case Marzo:
  case Noviembre:
    IncreDias = 3;
    break;
  case Junio:
    IncreDias = 4;
    break;
  case Septiembre:
  case Diciembre:
    IncreDias = 5;
    break;
  default:
    IncreDias = 6;
}
```

Evidentemente esta sentencia da lugar a un fragmento de programa más corto y fácil de entender.

En la sentencia **switch** se deben incluir todos los posibles valores que pueda tomar la variable o expresión. Cuando se obtiene un valor que no está asociado a ninguna vía (y no hay alternativa **default**), el programa finaliza por error. Si lo que sucede es que existen valores para los que no se debe realizar ninguna acción, entonces estos valores se deben declarar asociados a una acción vacía. Por ejemplo:

```
switch (M) {
  case Enero:
  case Febrero:
  case Marzo:
  case Abril:
  case Mayo:
  case Julio:
  case Agosto:
  case Septiembre:
  case Octubre:
  case Noviembre:
    break;
  case Junio:
  case Diciembre:
    sueldo = sueldo + extra;
    break;
}
```

Otra forma de conseguir esto mismo es mediante una alternativa **default** vacía utilizando sólo un punto y coma (;). Por ejemplo:

```
switch (M) {
  case Junio:
  case Diciembre:
    sueldo = sueldo + extra;
    break;
  default:
    ;
}
```

La diferencia entre ambas es que en el primer caso se produciría un error que finalizaría la ejecución del programa si por cualquier causa la variable M toma un valor fuera del rango de meses declarado de `Enero ... Diciembre`. En el segundo caso no se distingue la situación errónea de la que no lo es. Si el programa está completamente probado, es muy probable que tal situación no se produzca nunca. Sin embargo, cuando un programa está todavía en fase

de prueba es importante conocer todos los errores para analizar sus causas y corregirlos.

Para acabar este apartado se detalla la sintaxis formal de la sentencia `switch` de **C±**:

*Sentencia_SWITCH* **::=** `switch ( ` *Expresión* ` ) {` *Lista_de_casos* `}`

*Lista_de_casos* **::=** *Caso* **{** *Caso* **}**
    **[** `default:` *Secuencia_de_sentencias* **]**

*Caso* **::=** *Lista_de_opciones* *Secuencia_de_sentencias* `break ;`

*Lista_de_opciones* **::=** `case` *Valor* `:` **{** `case` *Valor* `:` **}**

El tipo del resultado de la *Expresión* y cada *Valor* de la lista deben ser compatibles.

## 10.3 Equivalencia entre estructuras

Como ya ha sido explicado, las estructuras básicas estrictamente necesarias para la *programación estructurada* son la selección entre dos alternativas `if` y la iteración `while`. Éstas se pueden considerar las estructuras primarias. El resto, que denominaremos estructuras secundarias, son en general más complejas y tienen como objetivo lograr programas más sencillos en situaciones particulares.

Cualquiera de las estructuras secundarias siempre puede ser expresada en función de las primarias. Sin embargo, no siempre es posible expresar una sentencia primaria en función de una secundaria. Por ejemplo, no se puede realizar una iteración `while` condicionada por más de una variable de tipo simple mediante una estructura `for` de **C±**. Tampoco es posible realizar una selección `if` condicionada por los valores que toma una expresión real mediante una estructura `switch`.

Aunque a lo largo de las explicaciones del tema 5 y en este mismo tema se han ido mostrando ciertas transformaciones y equivalencias entre estructuras, en este apartado a modo de resumen se muestra cómo cualquier estructura secundaria se puede realizar mediante las estructuras primarias.

### 10.3.1 Selección por casos

Esta estructura también se puede realizar mediante selecciones en cascada. Dada la estructura siguiente:

```
switch (Expresión) {
  case v1 :
    SentenciasA
    break;
  case v2 :
  case v3 :
  case v4 :
    SentenciasB
    break;
  case v5 :
  case v6 :
    SentenciasC
    break;
  . . . .
  default
    SentenciasH
}
```

se puede realizar de la siguiente forma:

```
valor = Expresion;
if (valor == v1) {
  SentenciasA
} else if ((valor == v2) || (valor == v3) || (valor == v4)) {
  SentenciasB
} else if ((valor == v5) || (valor == v6) {
  SentenciasC
. . . .
} else {
  SentenciasH
}
```

Esta construcción se puede poner a su vez en función de la estructura básica **if**, de forma anidada.

### 10.3.2 Bucle con contador

Esta estructura se puede hacer mediante un control explícito del contador del bucle. Sea la estructura siguiente:

```
for (int indice = Inicial; indice <= Final; indice++) {
  Sentencias
}
```

Cuando el incremento es positivo se puede realizar de la siguiente forma:

```
int indice = Inicial;
while (indice <= Final) {
  Sentencias
  indice++;
}
```

cuando el incremento es negativo la comparación se debe hacer por mayor o igual en lugar de por menor o igual.

### 10.3.3 Repetición

La sentencia `do` se puede transformar en `while` forzando la ejecución incondicional de la primera iteración. La estructura:

```
do {
  Sentencias
while (Condición);
```

se puede convertir en esta otra:

```
Sentencias
while (Condición) {
  Sentencias
}
```

O bien, usando una variable auxiliar para almacenar el valor de la condición:

```
seguir = true;
while (seguir) {
  Sentencias
  seguir = Condición;
}
```

## 10.4 Ejemplos de programas

En este apartado se recogen algunos programas completos que utilizan las estructuras estudiadas. Algunas partes de estos programas ya han sido utilizadas en los ejemplos presentados a lo largo de este tema.

### 10.4.1 Ejemplo: Imprimir tickets de comedor

Se trata de confeccionar un ticket mensual de comedor con la cantidad a pagar dependiendo de los días laborables de cada mes y descontando las ausencias

de cada persona. El objetivo fundamental es ilustrar el uso de la selección por casos en función del tipo enumerado de los meses del año. La selección del mes se realiza mediante una sentencia de repetición que controla que el mes este comprendido entre 1 y 12. Asimismo, el programa permite confeccionar todos los tickets necesarios hasta indicar que no se quiere continuar mediante un bucle repetitivo controlado por una respuesta del tipo Si/No. El listado completo del programa es el siguiente:

```
/********************************************
* Programa: Comedor
*
* Descripción:
*   Programa para realizar el ticket
*   de pago de un comedor
********************************************/
#include <stdio.h>

int main() {
  const float Menu    = 8.5;  /* precio del menú */
  const float IVAMenu = 7.0;  /* 7% IVA del menú */

  typedef enum TipoMes {
    Enero, Febrero, Marzo,
    Abril, Mayo, Junio,
    Julio, Agosto, Septiembre,
    Octubre, Noviembre, Diciembre
  };

  TipoMes mes;          /* mes a pagar */
  int dias, diasPagar, diasFaltas, aux;
  float total, totalIVA;
  char tecla;

  /*-- Leer y validar el mes leído --*/
  do {
    printf( "¿Mes? " );
    scanf( "%d", &aux );
  } while ((aux < 1) || (aux > 12));
  mes = TipoMes( aux-1 );

  /*-- Calcular días según el mes --*/
  switch (mes) {
    case Agosto:
      dias = 0;
      break;
```

```
    case Enero:
    case Abril:
    case Diciembre:
      dias = 17;
      break;
    case Febrero:
    case Septiembre:
      dias = 20;
      break;
    case Junio:
    case Noviembre:
      dias = 21;
      break;
    case Marzo:
    case Julio:
      dias = 22;
      break;
    case Mayo:
    case Octubre:
      dias = 23;
      break;
  }

  /*-- Confeccionar los tickets a partir de las ausencias --*/
  do {
    printf( "¿Total de Ausencias? " );
    scanf( "%d", &diasFaltas );
    printf( "----------------------------------\n" );

    diasPagar = dias - diasFaltas;
    total = diasPagar*Menu;
    totalIVA = total*IVAMenu/100.0;
    printf( "     RECIBO de COMEDOR\n" );
    printf( "Comidas   Precio       Total\n" );
    printf( "%4d%11.2f%13.2f\n", diasPagar, Menu, total );
    printf( "          %3.1f %% IVA   %6.2f\n", IVAMenu, totalIVA );
    printf( "       Total Recibo %8.2f Euros\n", total+totalIVA );
    printf( "----------------------------------\n" );

    tecla = ' ';
    printf( "¿Otro Recibo(S/N)? " );
    while ((tecla != 'S') && (tecla != 'N')) {
      scanf( "%c", &tecla );
    }
  } while (tecla != 'N');
}
```

Un ejemplo del resultado de la ejecución del programa es el siguiente:

```
¿Mes? 4
¿Total de Ausencias? 5
----------------------------------
     RECIBO de COMEDOR
Comidas   Precio       Total
  12       8.50       102.00
          7.0 % IVA     7.14
       Total Recibo   109.14 Euros
----------------------------------
¿Otro Recibo(S/N)? S
¿Total de Ausencias? 7
----------------------------------
     RECIBO de COMEDOR
Comidas   Precio       Total
  10       8.50       85.00
          7.0 % IVA     5.95
       Total Recibo    90.95 Euros
----------------------------------
¿Otro Recibo(S/N)? N
```

### 10.4.2 Ejemplo: Gestión de tarjetas de embarque

Este programa realiza la gestión automática de los asientos de un avión e imprime las tarjetas de embarque. Cuando varios pasajeros solicitan juntos las tarjetas de embarque se les asignan asientos de la misma fila. Si no existen asientos de la misma fila se ocupan los huecos libres. El número de tarjetas de embarque que se pueden solicitar simultáneamente es como máximo de 6. Los asientos se gestionan como un vector de filas al que inicialmente se le asignan un determinado número de asientos por fila.

El procedimiento **BuscarPlazas** es el encargado de realizar el algoritmo para situar las plazas solicitadas. En un primer intento se trata de asignar las nuevas plazas en la misma fila. Como segundo intento se asignarán las plazas en filas distintas. Cuando no hay suficientes plazas disponibles se dará un mensaje. Este procedimiento actualizará las plazas disponibles y el número total de plazas libres después de la asignación. Otro procedimiento es el encargado de imprimir las tarjetas de embarque cuando se ha encontrado sitio.

Para comprobar la situación de los asientos libres se dispone de un procedimiento que imprime las plazas libres y las ocupadas. El programa consiste en un menú que utiliza el procedimiento para la búsqueda y el procedimiento

para mostrar la situación según elección del operador. Todo el programa está realizado en base a las constantes del aforo, y el tamaño de la fila. Cualquier tamaño de avión de un mismo formato se puede gestionar cambiando estas dos constantes. El listado completo del programa es el siguiente:

```
/****************************************************
* Programa: Mostrador
*
* Descripción:
*   Este programa confecciona tarjetas de embarque
*   y asigna plazas en la misma fila si es posible
****************************************************/
#include <stdio.h>

const int Aforo = 60;        /* total asientos */
const int AsientosFila = 6; /* asientos por fila */

const int Filas = Aforo/AsientosFila;
const int Pasillo = AsientosFila/2;

typedef int Plazas[Filas];

/** Mostrar ocupación del avión */
void PintarPlazas(const Plazas P) {
  for (int i = 0; i < Filas; i++) {
    for (int j = AsientosFila; j >= 1; j--) {
      if (j == Pasillo ) {
        printf( "   " );
      }
      if (j > P[i]) {
        printf( "  (*) ");
      } else {
        printf( "  ( ) ");
      }
    }
    printf( "\n" );
  }
}

/** Imprimir una "tarjeta de embarque" */
void ImprimirTarjeta( int fila, int asiento ) {
  printf( ".--------------------------.\n" );
  printf( "|    TARJETA DE EMBARQUE    |\n" );
  printf( "|  Fila :%3d", fila );
  printf( "  Asiento :%3d  |\n", asiento );
  printf( "'--------------------------'\n" );
```

```
}

/** Buscar plazas libres contiguas */
void BuscarPlazas( int nuevas, int & libres, Plazas sitios ) {
  int ind, nue, dispo, aux;

  if (nuevas <= libres) {
    /*-- Buscar plazas en la misma fila --*/
    ind = 0;
    nue = nuevas;
    while ((ind < Filas) && (nue > 0)) {
      dispo = sitios[ind];
      if (nue <= dispo) {
        aux = AsientosFila - dispo;
        for (int i = 1; i <= nue; i++) {
          ImprimirTarjeta( ind+1, aux+i );
        }
        sitios[ind] = sitios[ind] - nue;
        libres = libres - nue;
        nue = 0;
      }
      ind++;
    }

    /*-- Buscar plazas en cualquier fila --*/
    ind = 0;
    while (nue > 0) {
      dispo = sitios[ind];
      if (dispo > 0) {
        aux = AsientosFila - dispo;
        for (int i = 1; i <= dispo; i++) {
          ImprimirTarjeta( ind+1, aux+i );
        }
        sitios[ind] = sitios[ind] - dispo;
        libres = libres - dispo;
        nue = nue - dispo;
      }
      ind++;
    }
  } else {
    printf( "No hay plazas suficientes\n" );
  }
}
```

```
/** Programa principal */
int main() {
  Plazas pasaje;
  int sitiosLibres = Aforo;
  char tecla;
  int aux;

  for (int i = 0; i < Filas; i++) {
    pasaje[i] = AsientosFila;
  }
  do {
    printf( "\n¿Opción (Tarjetas, Pasaje, Fin)? " );
    scanf( " %c", &tecla );
    switch (tecla) {
      case 'T':
        do {
          printf( "¿Número de plazas (1 a 6)? " );
          scanf( "%d", &aux );
        } while (aux < 1 || aux > 6);
        BuscarPlazas( aux, sitiosLibres, pasaje );
        break;
      case 'P':
        PintarPlazas( pasaje );
        break;
      default:
        ;
    }
  } while (tecla != 'F');
}
```

A continuación se muestra un fragmento de la ejecución del programa:

```
...

¿Opción (Tarjetas, Pasaje, Fin)? P
  (*)   (*)   (*)     (*)   (*)   ( )
  (*)   (*)   (*)     ( )   ( )   ( )
  (*)   (*)   (*)     (*)   ( )   ( )
  ( )   ( )   ( )     ( )   ( )   ( )
  ( )   ( )   ( )     ( )   ( )   ( )
  ( )   ( )   ( )     ( )   ( )   ( )
  ( )   ( )   ( )     ( )   ( )   ( )
  ( )   ( )   ( )     ( )   ( )   ( )
  ( )   ( )   ( )     ( )   ( )   ( )
  ( )   ( )   ( )     ( )   ( )   ( )
```

```
¿Opción (Tarjetas, Pasaje, Fin)? T
¿Número de plazas (1 a 6)? 5
.--------------------------.
|    TARJETA DE EMBARQUE    |
|  Fila :  4  Asiento :  1  |
'--------------------------'
.--------------------------.
|    TARJETA DE EMBARQUE    |
|  Fila :  4  Asiento :  2  |
'--------------------------'
...
.--------------------------.
|    TARJETA DE EMBARQUE    |
|  Fila :  4  Asiento :  5  |
'--------------------------'

¿Opción (Tarjetas, Pasaje, Fin)? P
  (*)   (*)   (*)     (*)   (*)   ( )
  (*)   (*)   (*)     ( )   ( )   ( )
  (*)   (*)   (*)     (*)   ( )   ( )
  (*)   (*)   (*)     (*)   (*)   ( )
  ( )   ( )   ( )     ( )   ( )   ( )
  ( )   ( )   ( )     ( )   ( )   ( )
  ( )   ( )   ( )     ( )   ( )   ( )
  ( )   ( )   ( )     ( )   ( )   ( )
  ( )   ( )   ( )     ( )   ( )   ( )
  ( )   ( )   ( )     ( )   ( )   ( )

¿Opción (Tarjetas, Pasaje, Fin)? T
¿Número de plazas (1 a 6)? 2
.--------------------------.
|    TARJETA DE EMBARQUE    |
|  Fila :  2  Asiento :  4  |
'--------------------------'
.--------------------------.
|    TARJETA DE EMBARQUE    |
|  Fila :  2  Asiento :  5  |
'--------------------------'
```

```
Opción (Tarjetas, Pasaje, Fin)? P
(*)  (*)  (*)     (*)  (*)  ( )
(*)  (*)  (*)     (*)  (*)  ( )
(*)  (*)  (*)     (*)  ( )  ( )
(*)  (*)  (*)     (*)  (*)  ( )
( )  ( )  ( )     ( )  ( )  ( )
( )  ( )  ( )     ( )  ( )  ( )
( )  ( )  ( )     ( )  ( )  ( )
( )  ( )  ( )     ( )  ( )  ( )
( )  ( )  ( )     ( )  ( )  ( )
( )  ( )  ( )     ( )  ( )  ( )

Opción (Tarjetas, Pasaje, Fin)? F
```

### 10.4.3 Ejemplo: Calculadora

Este programa simula una pequeña calculadora que realiza las cuatro operaciones aritméticas básicas (suma, resta, multiplicación y división), tiene 4 posiciones de memoria (A, B, C y D) para guardar resultados intermedios y además indica los errores que se producen al escribir las operaciones a realizar. El programa es un ejemplo de combinación compleja de estructuras de control de diferentes clases: selección sencilla o múltiple, bucles normales o indefinidos, con salida intermedia, etc.

El programa lee como entrada líneas de texto con las operaciones a realizar. Cada línea de entrada puede contener una o varias expresiones terminadas por punto y coma (`;`). Cada expresión está formada por una secuencia de operandos y operadores, que se evalúan estrictamente de izquierda a derecha.

Un operando puede ser cualquiera de las 4 posiciones de memoria o un valor numérico. Los valores numéricos se indican mediante una secuencia de dígitos del 0 al 9 sin blancos ni ningún otro carácter entre ellos. Es decir, los operandos numéricos literales son valores enteros, pero los cálculos se realizan con valores reales, de manera que un cociente puede dar un resultado con parte fraccionaria.

Un operador puede ser el símbolo de una operación aritmética (+, -, *, /). Además el resultado en un punto cualquiera se puede guardar en una posición de memoria indicándolo mediante el signo igual (=) seguido de la posición (`A`, `B`, `C` o `D`).

El final del programa se indica usando un punto (`.`) al final de la última expresión, en lugar de un punto y coma.

Son ejemplos de expresiones válidas las siguientes:

```
1232 * 456 = A;
 A+34=B-5;   A/B + 7 ;
```

El procedimiento **LeerCaracter** lee el siguiente carácter de la entrada y hace eco por la salida. También permite releer el carácter anterior si no se ha procesado efectivamente, sino que sólo se ha leído para detectar el final de un valor numérico. Además detecta el final del texto de entrada, devolviendo un carácter nulo en ese caso.

El procedimiento **LeerSimbolo** es el encargado de convertir grupos de caracteres de entrada en el símbolo currespondiente. Si es un operando numérico calcula además su valor.

El procedimiento **LeerOperando** lee el siguiente símbolo, comprueba que es un operando numérico o referencia a memoria, y en ese último caso recupera el valor almacenado.

El programa se plantea como un bucle indefinido en el que se analizan y ejecutan las operaciones que se introducen hasta el punto final, o hasta agotar el texto de entrada. Se detectan los errores en los operandos, en los operadores y la falta de alguno de los símbolos necesarios. En caso de error se ignora el resto de la expresión y se continúa con la siguiente, si la hay. El listado completo es el siguiente:

```
/****************************************************
* Programa: Calculadora
*
* Descripción:
*   Este programa simula una calculadora con las
*   cuatro operaciones básicas (+,-,*,/) y cuatro
*   posiciones de memoria. Analiza y evalúa las
*   expresiones e indica los errores que contengan,
*   si los hay
****************************************************/
#include <stdio.h>
#include <stdlib.h>

typedef enum TipoSimbolo {
  Numero, Memoria, Operador, Terminador, Desconocido
};
char c = ' '; /* carácter leído */
bool releer;  /* ya leído de antemano */
TipoSimbolo simbolo; /* símbolo leído y */
float numero;        /* su valor numérico */
char caracter;       /* o de texto */
bool primero; /* primer símbolo de una operación */
```

```
typedef float TipoMemoria[4];
TipoMemoria memoriaABCD;  /* 4 memorias */

/*===================================
  Leer siguiente carácter y hacer eco
  Devuelve nulo al final del fichero
  ===================================*/
void LeerCaracter() {
  if (primero) {          /* leer saltando espacios */
    primero = false;
    releer = false;
    c = '\0';
    scanf( " %c", &c );
  } else if (releer) { /* ya estaba leído de antemano */
    releer = false;
    return;
  } else {                /* lectura normal */
    c = '\0';
    scanf( "%c", &c );
  }
  printf( "%c", c);      /* hacer eco */
}

/*===============================================
  Procedimiento para reconocer el siguiente símbolo
  y dejarlo en 'simbolo', 'numero' y 'caracter'
  ===============================================*/
void LeerSimbolo() {
  do {
    LeerCaracter();
  }  while (c == ' ');
  caracter = c;
  switch (c) {
    case '+':
    case '-':
    case '*':
    case '/':
    case '=':
      simbolo = Operador;
      break;
    case 'A':
    case 'B':
    case 'C':
    case 'D':
      simbolo = Memoria;
      break;
```

```
    case ';':
    case '.':
      simbolo = Terminador;
      break;
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
      simbolo = Numero;
      numero = 0.0;
      do {
        numero = 10.0*numero + float( int(c)-int('0') );
        LeerCaracter();
      } while ((c >= '0') && (c <= '9'));
      releer = true;
      break;
    case '\n':
    case '\r':
      simbolo = Terminador;
      printf( "\n** Error: expresión incompleta\n" );
      break;
    case '\0':
      simbolo = Terminador;
      printf( "\n** Error: fin de la entrada\n" );
      break;
    default:
      simbolo = Desconocido;
      caracter = '\0';
      printf( "\n** Error: símbolo no reconocido\n" );
  }
}
```

```
/*===========================================
  Reconocer y obtener un operando con su valor
  ===========================================*/
void LeerOperando( float & valor, bool & error ) {
  error = false;
  LeerSimbolo();
  if (simbolo == Numero) {
    valor = numero;
  } else if (simbolo == Memoria) {
    valor = memoriaABCD[int(caracter)-int('A')];
  } else {
    error = true;
    if (caracter > ' ') {
      printf( "\n** Error: se necesita operando\n" );
    }
  }
}

/*===========================================
  Programa principal
  ===========================================*/
int main() {
  float operando, resultado;
  char operador;
  bool seguir, error;

  for (int k=0; k<4; k++) {
    memoriaABCD[k] = 0.0;
  }

  /*-- Bucle de procesar expresiones --*/
  do {
    printf( "Cálculo: " );
    resultado = 0.0;

    /* Leer y procesar el primer operando */
    primero = true;
    LeerOperando( resultado, error );
```

```
/* Leer y procesar el resto de la expresión */
seguir = !error;
while (seguir) {  /* bucle indefinido */

  /*-- Leer operador o fin de expresión --*/
  LeerSimbolo();
  switch (simbolo) {

    case Operador:
      /*-- Operaciones aritméticas o asignación --*/
      operador = caracter;
      LeerOperando( operando, error );
      if (error) {
        seguir = false;
        continue;  /* se da por terminada la expresión */
      }
      switch (operador) {
        case '+':
          resultado = resultado + operando;
          break;
        case '-':
          resultado = resultado - operando;
          break;
        case '*':
          resultado = resultado * operando;
          break;
        case '/':
          resultado = resultado / operando;
          break;
        default:  /* guardar en memoria */
          if (simbolo == Memoria) {
            memoriaABCD[int(caracter)-int('A')] = resultado;
          } else {
            printf( "\n** Error: Se necesita A, B, C o D\n" );
            error = true;
            seguir = false;
            continue;  /* se da por terminada la expresión */
          }
      }
      break;
```

```
      case Numero:
      case Memoria:
        /*-- Operador no válido --*/
        printf( "\n** Error: se necesita operador\n" );
        error = true;
        break;

      default:
        /*-- Fin de la expresión --*/
        seguir = false;
        error = (caracter != ';' && caracter != '.');
    }
    seguir = seguir && !error;
  } /* fin del bucle del resto de la expresión */

  if (!error) {
    /*-- Escribir resultado --*/
    printf( "\n>> Resultado: %g\n", resultado );
  } else if (c!='\0') {
    /*-- Saltar hasta fin de expresión, de línea o de fichero --*/
    printf( "<< " );
    while (c!=';' && c!='.' && c!='\n' && c!='\0') {
      LeerCaracter();
    }
    printf( "\n" );
  }
  printf( "\n" );
 } while (c != '.' && c != '\0'); /* fin del bucle principal */
}
```

A continuación se muestra un ejemplo de ejecución. El texto de entrada contiene las siguientes expresiones:

```
22 + 33 = A - 25 = B; A; B*2;
3; 8 * (5 - A);  44
A = 33 + 7;      9 / 7 + 5;
+22;  final.
```

Y el resultado obtenido es:

```
Cálculo: 22 + 33 = A - 25 = B;
>> Resultado: 30

Cálculo: A;
>> Resultado: 55

Cálculo: B*2;
>> Resultado: 60

Cálculo: 3;
>> Resultado: 3

Cálculo: 8 * (
** Error: símbolo no reconocido
<< 5 - A);

Cálculo: 44
** Error: expresión incompleta
<<


Cálculo: A = 33
** Error: Se necesita A, B, C o D
<< + 7;

Cálculo: 9 / 7 + 5;
>> Resultado: 6.28571

Cálculo: +
** Error: se necesita operando
<< 22;

Cálculo: f
** Error: símbolo no reconocido
<< inal.
```

# Ejercicios sin resolver - II

A continuación se enuncian un segundo bloque de ejercicios sin resolver. Todos ellos pueden y deben ser realizados en **C±** utilizando las herramientas y metodología explicada hasta este momento. Los enunciados de los ejercicios son los siguientes:

1. Realizar una función que devuelva un valor booleano cierto o falso si el número que se le pasa como argumento es "perfecto". Para que un número sea "perfecto" es necesario que su valor sea igual a la suma de todos sus divisores incluyendo al 1 y sin incluirle a él mismo. Por ejemplo, los divisores de 6 son el 1, el 2 y el 3; su suma es igual a 6, luego el número 6 es "perfecto". Utilizando la función anterior realizar un programa que escriba la lista de números "perfectos" hasta uno dado, introducido como dato al programa.
2. Realizar un procedimiento que lea una letra, compruebe que es una de las utilizadas para escribir los números romanos y devuelva su valor, según la siguiente tabla:

   | | | | | | |
   |---|---|---|---|---|---|
   | I | - 1 | | C | - | 100 |
   | V | - | 5 | D | - | 500 |
   | X | - | 10 | M | - | 1000 |
   | L | - | 50 | Resto | - | 0 |

   Utilizando el procedimiento anterior, realizar otro procedimiento que lea un número romano y devuelva su valor entero correspondiente. Este procedimiento tendrá en cuenta las reglas de escritura de los números romanos. Finalmente, utilizando este último procedimiento realizar un programa que lea dos números romanos e indique cuál de ellos es mayor.
3. Realizar un programa que escriba todas las permutaciones posibles que se pueden obtener a partir de las 4 letras: A, B, C y D.
4. Realizar una función que devuelva el día de la semana cuando se le pasan como argumentos el día, el mes y el año de una fecha cualquiera. Utilizando la función anterior escribir un programa al que se le introdu-

cen como datos el mes y el año y devuelve como resultado la hoja del calendario de dicho mes. Por ejemplo:

```
Mes? 1
Año? 2018

          ENERO 2018
LU  MA  MI  JU  VI  SA  DO
 1   2   3   4   5   6   7
 8   9  10  11  12  13  14
15  16  17  18  19  20  21
22  23  24  25  26  27  28
29  30  31
```

5. Realizar un programa que analice un texto terminado con un punto (.) y elabore las siguientes estadísticas:
   - Número total de palabras del texto
   - Número de palabras que utilizan N o más vocales diferentes
   - Número de palabras que utilizan M o más consonantes diferentes.

   Los valores de N y M se leerán como datos del programa.
6. Realizar un programa para controlar las plazas de un aparcamiento. El aparcamiento dispone de 25 plazas de dos tamaños diferentes: 15 pequeñas y 10 grandes con la disposición que se muestra a continuación:

| G 1 | G 2 | G 3 | G 4 | G 5 | G 6 | G 7 | G 8 | G 9 | G 10 |
|---|---|---|---|---|---|---|---|---|---|

| P 1 | P 2 | P 3 | P 4 | P 5 | P 6 | P 7 | P 8 | P 9 | P 10 | P 11 | P 12 | P 13 | P 14 | P 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

   La asignación se realizará automáticamente según el tamaño del vehículo que se quiere aparcar con el siguiente algoritmo:
   - Cada vehículo solamente ocupará una plaza.
   - Un vehículo pequeño siempre ocupará una plaza pequeña, salvo que estén todas ocupadas y exista alguna grande libre.
   - Un vehículo grande sólo puede aparcar en una plaza grande. Si todas están ocupadas no podrá aparcar aunque estén todas las pequeñas libres.
   - De todas las plazas libres, siempre se ocupará primero la de número menor.

El programa tendrá 3 opciones básicas:

- Entrada: es necesario indicar el tamaño de coche ( P/G ).
- Salida: es necesario indicar la plaza que se deja libre. Por ejemplo: P 5.
- Situación del aparcamiento: indicando las plazas libres y las ocupadas.

# Tema 11

# Estructuras de datos

En el tema 9 se han introducido mecanismos básicos para la definición de nuevos tipos, incluyendo tipos estructurados: formaciones o vectores (*arrays*) y tuplas (`struct`). En este tema se introducen nuevas estructuras de datos que aportan posibilidades adicionales.

En particular se generaliza la estructura vectorial mediante el uso de vectores abiertos y matrices o formaciones de varias dimensiones, así como los tipos unión, conjuntos, y estructuras de datos complejas que combinan varias estructuras más sencillas.

## 11.1 Argumentos de tipo vector abierto

Si un subprograma debe operar con un vector recibido como argumento, necesita toda la información del tipo de dicho vector, es decir, el tipo y número de sus elementos. Por ejemplo, en el tema 9 aparecían los siguientes fragmentos de código:

```
const int NumeroElementos = 10;
typedef int TipoVector[NumeroElementos];

void EscribirVector( const TipoVector v ) {...}

TipoVector vectorUno, vectorDos;

EscribirVector( vectorDos );
```

El código del procedimiento `EscribirVector` puede redactarse con seguridad, ya que se conoce toda la información del tipo del argumento:

```
void EscribirVector( const TipoVector v ) {
  for (int i = 0; i < NumeroElementos; i++) {
    printf( "%10d", v[i] );
  }
  printf( "\n" );
}
```

Si ahora necesitamos trabajar con vectores de otro tamaño, tendremos que definir un nuevo tipo y, lamentablemente, también un nuevo procedimiento de escritura (recordemos que en **C±** el argumento real debe ser exactamente del mismo tipo que el argumento formal):

```
const int NumeroElementosLargo = 100;
typedef int TipoVectorLargo[NumeroElementosLargo];

void EscribirVectorLargo( const TipoVectorLargo v ) {
  for (int i = 0; i < NumeroElementosLargo; i++) {
    printf( "%10d", v[i] );
  }
  printf( "\n" );
}
```

Como vemos, el código del nuevo procedimiento de escritura es idéntico al del anterior, salvo por la referencia al número de elementos. En la práctica no tiene sentido tener que duplicar el código sólo porque cambia el valor de una constante. Más razonable es escribir un procedimiento general de escritura de vectores de números enteros, y pasar como parámetro el tamaño del vector. Para eso hace falta un mecanismo para expresar que un argumento de tipo vector puede tener un tamaño cualquiera, es decir, indefinido. Los vectores con un tamaño indefinido se denominan *vectores abiertos*.

En **C±** los argumentos de tipo vector abierto se especifican de manera similar a una declaración de tipo vector, omitiendo el tamaño explícito pero no los corchetes (`[]`). Ejemplo:

```
void EscribirVectorAbierto( const int v[], int numElementos ) {
  for (int i = 0; i < numElementos; i++) {
    printf( "%10d", v[i] );
  }
  printf( "\n" );
}
```

Ahora podemos usar este procedimiento para escribir vectores de cualquiera de los tipos anteriores:

```
TipoVector vectorUno, vectorDos;
TipoVectorLargo vectorLargo;

EscribirVectorAbierto( vectorDos, NumeroElementos );
EscribirVectorAbierto( vectorLargo, NumeroElementosLargo );
```

El precio que hay que pagar por disponer de esta facilidad es tener que pasar siempre la longitud concreta del vector como argumento, en cada llamada.

Por supuesto, hay otras alternativas al paso explícito de la longitud del vector. Los subprogramas estándar para operar con cadenas de caracteres (*strings*) usan la técnica de almacenar un carácter nulo al final del valor efectivo de cada cadena. Esto exige disponer siempre de espacio para un carácter más, al menos, pero evitan tener que pasar la longitud como argumento. A continuación se muestra un ejemplo de esta técnica.

### 11.1.1 Ejemplo: Contar letras y dígitos

Este ejemplo analiza un fragmento de texto y cuenta el número de caracteres, letras, dígitos y espacios en blanco de los que consta. Se usa un subprograma que recibe el fragmento de texto como vector abierto, y analiza uno a uno sus caracteres hasta encontrar el carácter nulo que marca el final. El listado completo del programa es el siguiente:

```
/************************************************
* Programa: ContarLetrasyDigitos
*
* Descripción:
*   Programa que analiza fragmentos de texto y
*   cuenta las letras y dígitos lo componen
************************************************/
#include <stdio.h>
#include <ctype.h>

/** Analizar el texto */
void AnalizarTexto( const char texto[] ) {
  int letras = 0;
  int digitos = 0;
  int blancos = 0;
  int posi = 0;
```

```
  while (texto[posi] != '\0') {
    if (isalpha(texto[posi])) {
      letras++;
    } else if (isdigit(texto[posi])) {
      digitos++;
    } else if (isspace(texto[posi])) {
      blancos++;
    }
    posi++;
  }

  printf( "Texto: %s\n", texto );
  printf(
    "Longitud: %2d  Letras: %2d  Digitos: %2d  Blancos: %2d\n\n",
    posi, letras, digitos, blancos );
}

/** Programa principal */
int main() {
  AnalizarTexto( "12 de Octubre de 1492" );
  AnalizarTexto( "2001, una odisea del espacio" );
  AnalizarTexto( "" );
}
```

y el resultado de la ejecución es el siguiente:

```
Texto: 12 de Octubre de 1492
Longitud: 21  Letras: 11  Digitos:  6  Blancos:  4

Texto: 2001, una odisea del espacio
Longitud: 28  Letras: 19  Digitos:  4  Blancos:  4

Texto:
Longitud:  0  Letras:  0  Digitos:  0  Blancos:  0
```

## 11.2 Formaciones anidadas. Matrices

En esta sección se generaliza el concepto de vector para disponer de formaciones de más de una dimensión. Para simplificar, la exposición se centra en *matrices* de dos dimensiones, pero la notación y técnicas de uso se pueden extender a formaciones de más de dos dimensiones.

### 11.2.1 Declaración de matrices y uso de sus elementos

Las matrices son estructuras de tipo *formación* (*array*) de dos o más dimensiones. Una forma sencilla de plantear la definición de estas estructuras es considerarlas como vectores cuyos elementos son a su vez vectores (o matrices). La figura 11.1 muestra la estructura de una matriz de dos dimensiones, formada por filas o columnas.

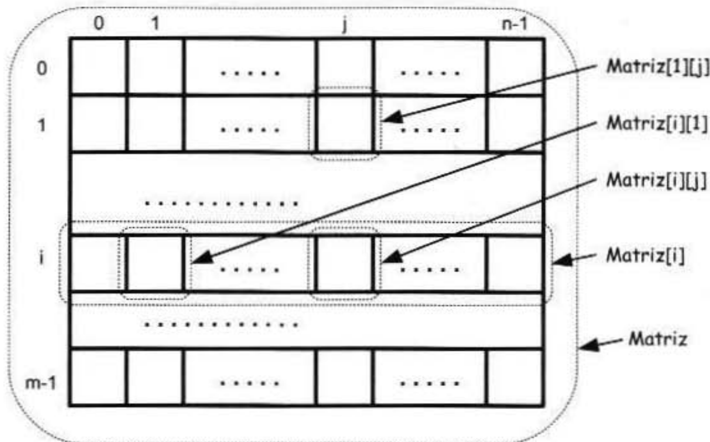



Figura 11.1 Matriz de dos dimensiones.

En la figura se presenta la matriz como un vector cuyos elementos son las filas de la matriz, que a su vez son vectores de elementos de la matriz. Usando directamente declaraciones de tipos vector podríamos escribir la declaración de una matriz de números enteros de la siguiente manera:

```
const int NumFilas = 10;
const int NumColumnas = 15;
typedef int TipoElemento;

typedef TipoElemento TipoFila[NumColumnas];
typedef TipoFila TipoMatriz[NumFilas];
```

Para designar un elemento de la matriz se usará un doble subíndice:

```
TipoMatriz matriz;

matriz[3][5] = 27;
```

La declaración de una matriz puede simplificarse en cierta medida respecto a la notación anterior dado que no es necesario declarar un tipo intermedio para las filas de la matriz. Por ejemplo:

```
const int NumFilas = 10;
const int NumColumnas = 15;
typedef int TipoElemento;

typedef TipoElemento TipoMatriz[NumFilas][NumColumnas];

TipoMatriz matriz;
```

Como recordatorio, repetiremos aquí las reglas sintácticas de la definición de formaciones, de una o varias dimensiones.

*Tipo_formación* **::=** `typedef` *Identificador_de_tipo_elemento* *Identificador_de_tipo_formación Dimensiones* ;

*Dimensiones* **::=** *Tamaño* **{** *Tamaño* **}**

*Tamaño* **::=** [ *Número_de_elementos* ]

## 11.2.2 Operaciones con matrices

Las operaciones con elementos individuales de una matriz pueden hacerse directamente, de forma análoga a la operación con variables simples de ese tipo. En cambio las operaciones globales con matrices han de plantearse de manera similar a las operaciones globales con vectores. En general habrá que operar elemento a elemento, o a lo sumo por filas completas.

Por ejemplo, para imprimir matrices del tipo declarado anteriormente podríamos escribir:

```
void EscribirMatriz( const TipoMatriz m ) {
  for (int i = 0; i < NumFilas; i++) {
    for (int j = 0; j < NumColumnas; j++) {
      printf( "%10d", m[i][j] );
    }
    printf( "\n" );
  }
}
```

En este ejemplo se ha operado directamente con cada elemento de la matriz, usando un doble bucle para recorrerla. También es posible operar con la matriz por filas completas, y detallar por separado cómo se opera con cada fila. Por supuesto, esto requiere haber definido explícitamente el tipo de las filas:

```
void EscribirFila( const TipoFila f ) {
  for (int i = 0; i < NumColumnas; i++) {
    printf( "%10d", f[i] );
  }
  printf( "\n" );
}

void EscribirMatriz( const TipoMatriz m ) {
  for (int i = 0; i < NumFilas; i++) {
    EscribirFila( m[i] );
  }
}
```

Al comienzo de este tema se ha mencionado la posibilidad de definir argumentos de tipo vector como vectores abiertos, de tamaño indefinido. Podemos pensar en generalizar ese mecanismo y permitir el uso de argumentos de tipo matriz abierta. Lamentablemente los lenguajes C o C++, y por lo tanto C±, no lo permiten. El siguiente fragmento de código provoca un error de compilación:

```
void EscribirMatrizAbierta( const int m[][],          /* <-- ERROR */
                            int filas, int columnas )
```

Por supuesto, si se manipula una matriz como vector de vectores sí se puede usar el mecanismo de vector abierto, pero limitado a formaciones de una dimensión cuyos elementos tienen un tipo explícito de tamaño conocido. A continuación se repite el código de ejemplo anterior ligeramente modificado para usar en lo posible el mecanismo de vector abierto, aunque de hecho no aporta ninguna ventaja importante:

```
void EscribirFila( const TipoElemento f[], int columnas ) {
  for (int i = 0; i < columnas; i++) {
    printf( "%10d", f[i] );
  }
  printf( "\n" );
}

void EscribirMatriz( const TipoFila m[] ) {
  for (int i = 0; i < NumFilas; i++) {
    EscribirFila( m[i], NumColumnas );
  }
}
```

Si realmente se necesita definir operaciones genéricas con formaciones de dimensiones indefinidas hay que recurrir al uso explícito de punteros, tal como se verá en el tema 13.

## 11.2.3 Ejemplo: Contrastar una imagen

Las imágenes almacenadas por puntos (*pixels*) son un ejemplo típico de estructura matricial. Este programa trabaja con una imagen monocroma en la que cada punto puede tener un nivel de gris que varía de 0 a 5. La imagen se introduce como dato en forma de matriz de caracteres numéricos, con una fila por línea de texto, y a continuación se constrasta convirtiéndola a imagen con sólo niveles de blanco o negro puros. La imagen se presenta a la salida también como matriz de caracteres, usando caracteres apropiados para aproximar visualmente los distintos niveles de gris. El listado completo del programa es el siguiente:

```
/****************************************************
* Programa: Contrastar
*
* Descripción:
*   Este programa convierte una imagen digitalizada
*   a niveles de blanco y negro puros
****************************************************/
#include <stdio.h>

const int Ancho  = 40;  /* anchura de la imagen */
const int Alto   = 20;  /* altura de la imagen */
const int Blanco = 0;   /* nivel bajo de grises = blanco */
const int Negro  = 5;   /* nivel alto de grises = negro */

typedef int Imagen_t[Alto][Ancho];

/** Leer la imagen */
void LeerImagen( Imagen_t imagen ) {
  char c;

  for (int i=0; i<Alto; i++) {
    for (int j=0; j<Ancho; j++) {
      scanf( " %c", &c );
      imagen[i][j] = int(c) - int('0');
    }
  }
}
```

```
/** Contrastar la imagen */
void Contrastar( Imagen_t imagen, int nivel ) {
  for (int i=0; i<Alto; i++) {
    for (int j=0; j<Ancho; j++) {
      if (imagen[i][j] <= nivel) {
        imagen[i][j] = Blanco;
      } else {
        imagen[i][j] = Negro;
      }
    }
  }
}

/** Imprimir la imagen */
void Imprimir( const Imagen_t imagen ) {
  const char Punto[6] = {' ', '.', '+', 'x', '*', '#'};

  for (int i=0; i<Alto; i++) {
    for (int j=0; j<Ancho; j++) {
      printf( "%c", Punto[imagen[i][j]] );
    }
    printf( "\n" );
  }
}

/** Programa principal */
int main() {
  Imagen_t imagen;

  LeerImagen( imagen );
  printf( "Imagen inicial:\n" );
  Imprimir( imagen );

  Contrastar( imagen, 3 );
  printf( "\nImagen contrastada:\n" );
  Imprimir( imagen );
}
```

A continuación se presenta un ejemplo de la ejecución de este programa. Los datos de entrada contienen un carácter por cada punto de la imagen, y simulan una rejilla. Los datos son:

```
0123454321012345432101234543210123454321
1234545432123454543212345454321234545432
2345434543234543454323454345432345434543
3454323454345432345434543234543454323454
4543212345454321234545432123454543212344
5432101234543210123454321012345432101233
4543212345454321234545432123454543212344
3454323454345432345434543234543454323454
2345434543234543454323454345432345434543
1234545432123454543212345454321234545432
0123454321012345432101234543210123454321
1234545432123454543212345454321234545432
2345434543234543454323454345432345434543
3454323454345432345434543234543454323454
4543212345454321234545432123454543212345
5432101234543210123454321012345432101234
4543212345454321234545432123454543212345
3454323454345432345434543234543454323454
2345434543234543454323454345432345434543
1234545432123454543212345454321234545432
```

El resultado de la ejecución del programa es el siguiente:

```
Imagen inicial:
 .+x*#*x+.  .+x*#*x+.  .+x*#*x+.  .+x*#*x+.
.+x*#*#*x+.+x*#*#*x+.+x*#*#*x+.+x*#*#*x+
+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x
x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*
*#*x+.+x*#*#*x+.+x*#*#*x+.+x*#*#*x+.+x**
#*x+.  .+x*#*x+.  .+x*#*x+.  .+x*#*x+.  .+xx
*#*x+.+x*#*#*x+.+x*#*#*x+.+x*#*#*x+.+x**
x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*
+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x
.+x*#*#*x+.+x*#*#*x+.+x*#*#*x+.+x*#*#*x+
 .+x*#*x+.  .+x*#*x+.  .+x*#*x+.  .+x*#*x+.
.+x*#*#*x+.+x*#*#*x+.+x*#*#*x+.+x*#*#*x+
+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x
x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*
*#*x+.+x*#*#*x+.+x*#*#*x+.+x*#*#*x+.+x*#
#*x+.  .+x*#*x+.  .+x*#*x+.  .+x*#*x+.  .+x*
*#*x+.+x*#*#*x+.+x*#*#*x+.+x*#*#*x+.+x*#
x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*
+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x
.+x*#*#*x+.+x*#*#*x+.+x*#*#*x+.+x*#*#*x+
```

```
Imagen contrastada:
    ###       ###       ###       ###
   #####     #####     #####     #####
  ### ###   ### ###   ### ###   ### ###
 ###   ### ###   ### ###   ### ###   ###
###     #####     #####     #####     ##
##       ###       ###       ###
###     #####     #####     #####     ##
 ###   ### ###   ### ###   ### ###   ###
  ### ###   ### ###   ### ###   ### ###
   #####     #####     #####     #####
    ###       ###       ###       ###
   #####     #####     #####     #####
  ### ###   ### ###   ### ###   ### ###
 ###   ### ###   ### ###   ### ###   ###
###     #####     #####     #####     ##
##       ###       ###       ###       #
###     #####     #####     #####     ##
 ###   ### ###   ### ###   ### ###   ###
  ### ###   ### ###   ### ###   ### ###
   #####     #####     #####     #####
```

## 11.3 El esquema unión

Hay aplicaciones en las que resultaría deseable que el tipo de un dato variase según las circunstancias. Si las posibilidades de variación son un conjunto finito de tipos, entonces se puede decir que el tipo del dato corresponde a un esquema que es la *unión* de los tipos particulares posibles. Cada uno de los tipos particulares constituye una *variante* o alternativa del tipo unión. Representaremos simbólicamente este esquema de forma similar a las alternativas en las reglas de sintaxis BNF:

tipo_unión **=** variante **|** variante ...

Como situaciones típicas en las que se pueden aplicar los esquemas unión tenemos, entre otras, las siguientes:

- Datos que pueden representarse de diferentes maneras.
- Programas que operan indistintamente con varias clases de datos.
- Datos estructurados con elementos opcionales.

Algunos ejemplos concretos serían:

número_general **=** entero **|** fracción **|** real

coordenadas = coordenadas_cartesianas | coordenadas_polares

figura = punto | círculo | cuadrado | rectángulo | rombo | triángulo | elipse

datos_persona = datos_soltero | datos_menor | datos_casado

El primer caso correspondería a un programa que opere indistintamente con números de diferentes clases. Los números enteros son datos simples, al igual que los reales, aunque las colecciones de valores son diferentes. Los números fraccionarios se representan como dos números enteros (numerador y denominador).

En el segundo caso tenemos dos sistemas de coordenadas diferentes para representar los puntos del plano. Las coordenadas cartesianas son dos longitudes, mientras que las polares son una longitud y un ángulo.

En el tercer caso tenemos un programa que maneja un repertorio limitado de elementos gráficos. Para cada figura se necesitará conocer una colección de parámetros diferentes.

En el último caso, los datos de un soltero mayor de edad constituirían la información básica de una persona. Los menores deberían tener además un tutor o persona responsable de ellos, y los casados deberían tener una fecha de matrimonio y un cónyuge.

### 11.3.1 El tipo union

Los esquemas unión pueden utilizarse en programas en **C±** definiéndolos como tipos `union`. Un tipo `union` se define como una colección de campos alternativos, de tal manera que cada dato particular sólo usará uno de esos campos en un momento dado, dependiendo de la alternativa aplicable. La definición es similar a la de un agregado o `struct`, usando ahora la palabra clave `union`. Por ejemplo:

```
typedef struct TipoFraccion {
  int numerador;
  int denominador;
};

typedef union TipoNumero {
  int valorEntero;
  float valorReal;
  TipoFraccion valorRacional;
};
```

La referencia a los elementos componentes se hace también como en los tipos struct:

```
TipoNumero numero, otro, fraccion1, fraccion2;

numero.valorEntero = 33;
otro.valorReal = float(numero.valorEntero);
fraccion2.valorRacional = fraccion1.valorRacional;
```

Si una variante de una unión es a su vez otra tupla o unión habrá que usar varios cualificadores para designar los campos anidados:

```
fraccion1.valorRacional.numerador = 33;
fraccion1.valorRacional.denominador = 44;
```

Como se ha dicho, sólo una de las variantes puede estar vigente en un momento dado. Si asignamos valor a una de ellas será ésta la que exista a partir de ese momento, al tiempo que dejan de existir las demás:

```
fraccion1.valorEntero = 22;
printf( "%d", fraccion1.valorEntero );                  /* Correcto */
printf( "%d", fraccion1.valorRacional.denominador ); /* ERROR */
```

Es fácil darse cuenta de que un dato de un tipo unión no contiene en sí mismo información de cuál es la variante activa en un momento dado. Dicha información debe estar disponible de forma clara por otros medios. Si no es así, será muy fácil que se produzcan errores. Por ejemplo, al redactar un procedimiento para imprimir un dato del TipoNumero deberíamos pasar también como argumento una indicación de la variante aplicable:

```
typedef enum ClaseNumero {Entero, Real, Fraccion};

void EscribirNumero( TipoNumero n, ClaseNumero c ) {
  switch (c) {
  case Entero:
    printf( "%d", n.valorEntero );
    break;
  case Real:
    printf( "%f", n.valorReal );
    break;
  case Fraccion:
    printf( "%d/%d", n.valorRacional.numerador, n.valorRacional.denominador );
    break;
  default:
    printf( "????" );
  }
}
```

### 11.3.2 Registros con variantes

El hecho de que un dato de tipo unión deba ir acompañado de información complementaria para saber cuál es la variante aplicable hace que los tipos unión aparezcan casi siempre formando parte de estructuras más complejas.

Un ejemplo frecuente es lo que se denominan *registros con variantes.* Se trata de agregados o tuplas en los que hay una colección de campos fijos, aplicables en todos los casos, y campos variantes que se definen según el esquema unión. Además suele reservarse un campo fijo para indicar explícitamente cuál es la variante aplicable en cada momento. A dicho campo se le llama *discriminante.*

A continuación se recodifica el ejemplo anterior usando esta técnica:

```
typedef enum ClaseNumero {Entero, Real, Fraccion};

typedef struct TipoFraccion {
  int numerador;
  int denominador;
};

typedef union TipoValor {
  int valorEntero;
  float valorReal;
  TipoFraccion valorRacional;
};

typedef struct TipoNumero {
  ClaseNumero clase;  /* discriminante */
  TipoValor valor;
};

void EscribirNumero( TipoNumero n ) {
  switch (n.clase) {
  case Entero:
    printf( "%d", n.valor.valorEntero );
    break;
  case Real:
    printf( "%f", n.valor.valorReal );
    break;
  case Fraccion:
    printf( "%d/%d", n.valor.valorRacional.numerador,
                     n.valor.valorRacional.denominador );
    break;
  default:
    printf( "????" );
  }
```

```
}
```

Si el tratamiento de los registros con variantes se hace sólo mediante subprogramas que comprueban siempre el discriminante antes de operar, el código resulta mucho más seguro. Esta garantía de seguridad compensa perfectamente el esfuerzo adicional que representa definir y usar estructuras anidadas, como en este ejemplo. Además el uso de los subprogramas de manipulación es ahora más sencillo, ya que basta pasar como argumento sólo el propio dato, que tiene en sí mismo toda la información necesaria.

## 11.4 Esquemas de datos y esquemas de acciones

Una vez presentadas las estructuras de datos de tipos formación y registro, incluidos los registros con variantes, es interesante hacer notar la analogía que puede establecerse entre los esquemas de acciones y los esquemas de datos.

La programación estructurada recomienda usar los esquemas más sencillos posibles para organizar un elemento complejo de un programa a partir de elementos más simples. Recordaremos que estos esquemas básicos eran la *secuencia*, la *selección*, y la *iteración*, en el caso de acciones compuestas.

Estos esquemas se corresponden, respectivamente, con los esquemas *tupla*, *unión* y *formación*, definidos para las estructuras de datos. La analogía se pone de manifiesto si describimos dichos esquemas de forma generalizada, común a los datos y acciones.

*Tupla - Secuencia*: Colección de elementos de tipos diferentes, combinados en un orden fijo.

*Unión - Selección*: Selección de un elemento entre varios posibles, de tipos diferentes.

*Formación - Iteración*: Colección de elementos del mismo tipo.

Todavía puede manifestarse la analogía con más intensidad si observamos que el tratamiento de las componentes de una estructura de datos se realiza fácilmente con una acción compuesta de tipo análogo. Por ejemplo:

```
/* ESQUEMAS DE DATOS */

/* Esquema Tupla */
  typedef struct TipoTupla {
    char uno, dos, tres;
  };
  TipoTupla agregado;
```

```
/* Esquema Unión */
  typedef union TipoUnion {
    int alfa;
    float beta;
  };
  TipoUnion variante;
  bool numeroEntero;

/* Esquema Formación */
  typedef int TipoFormacion[10];
  TipoFormacion vector;

/* ESQUEMAS DE ACCIONES */

/* Secuencia para imprmir los campos una Tupla */
  printf( "%c", agregado.uno );
  printf( "%c", agregado.dos );
  printf( "%c", agregado.tres );

/* Selección para imprimir las variantes de una Union */
  if (numeroEntero) {
    printf( "%d", variante.alfa );
  } else {
    printf( "%f", variante.beta );
  }

/* Iteración para imprimir una Formación */
  for (int k=0; k<10; k++) {
    printf( "%d", vector[k] );
  }
```

Para imprimir el contenido de cada estructura de datos se ha utilizado precisamente la acción compuesta de estructura análoga.

## 11.5 Estructuras combinadas

Como en cualquier otro lenguaje de programación actual, en **C±** se pueden combinar entre sí los esquemas tupla, unión y formación para definir estructuras de datos complejas, exactamente igual a como se combinan la secuencia, selección e iteración para construir el código de acciones complejas. De hecho ya se ha hecho así en algún ejemplo anterior.

Una característica de los lenguajes de programación modernos es que se pueden combinar con bastante libertad elementos de la misma naturaleza. Esto ocurre en **C±** con las sentencias estructuradas, que permiten anidar esquemas de tipo secuencia, selección e iteración unos dentro de otros.

Con las estructuras de datos ocurre algo similar. Se pueden definir estructuras cuyas componentes son a su vez estructuras, sin límite de complejidad de los esquemas de datos resultantes.

### 11.5.1 Formas de combinación

La manera de combinar las estructuras de datos es hacer que los elementos de una estructura sean a su vez otras estructuras, sin limitación en la profundidad de anidamiento. Podemos analizar algunos ejemplos:

```
typedef struct TipoPunto {
  float x, y;
};

typedef TipoPunto TipoTriangulo[3];

const int MaxPuntos = 100;
typedef TipoPunto TipoPuntos[MaxPuntos];

typedef struct TipoListaPuntos {
  int numeroPuntos;
  TipoPuntos puntos;
};

typedef TipoListaPuntos Poligonal; /* línea abierta */
typedef TipoListaPuntos Poligono;  /* línea cerrada */
```

En estos ejemplos tenemos la representación de un triángulo como una formación de tres puntos, que son registros. La lista de puntos se estructura como un registro, uno de cuyos campos es una formación de puntos, que son a su vez registros. Las últimas declaraciones definen tipos sinónimos de la lista de puntos, general, con nombres particulares para ser usados en casos particulares.

Los siguientes ejemplos definen registros que contienen campos que son cadenas de caracteres. El último ejemplo es un registro que contiene a su vez otro registro anidado.

```
typedef char TipoNombre[20];
typedef char TipoApellido[30];
```

```
typedef char TipoLineaTexto[40];
typedef enum TipoProvincia {
  SinProvincia, Alava, Albacete,
  ... Zaragoza
};
typedef struct TipoNombreCompleto {
  TipoNombre nombre;
  TipoApellido apellido1, apellido2;
};

typedef struct TipoDomicilio {
  TipoLineaTexto calle;
  int numero;
  TipoLineaTexto zona, poblacion;
  int codigoPostal;
  TipoProvincia provincia;
};

typedef struct TipoDatosPersonales {
  TipoNombreCompleto nombreyApellidos;
  TipoDomicilio domicilio;
};
```

Cuando se utilizan estructuras combinadas, para hacer referencia a una componente en particular hay que usar los selectores apropiados, encadenados uno tras otro. Como ejemplos de uso de las estructuras definidas en este apartado, podemos poner:

```
TipoTriangulo pieza;
TipoListaPuntos camino;
TipoDatosPersonales empleado;

pieza[0].x = 2.33;
pieza[0].y = -3.45;
pieza[1].x = pieza[1].y;
pieza[1].y = 88.3;

camino.numeroPuntos = 3;
camino.puntos[0].x = 5.67;
camino.puntos[0].y = 7.21;
camino.puntos[1] = pieza[2];
camino.puntos[2] = camino.puntos[1];

strcpy( empleado.nombreyApellidos.nombre, "Alberto");
printf( "%s", empleado.domicilio.calle );
if (empleado.domicilio.poblacion[0] == '\0') {
```

```
  printf( "Sin domicilio" );
}
```

En estos ejemplos aparecen combinados los selectores de campo de registro y de componente de formación. La combinación es admisible cuando una componente de una estructura es a su vez una estructura de datos. A cada estructura se debe aplicar el selector que le corresponda.

### 11.5.2 Tablas

Aunque el estudio de las estructuras de datos excede del ámbito de este libro, resulta interesante mencionar algunos esquemas típicos que se obtienen combinando estructuras básicas. Este es el caso del esquema de *tabla*, que puede plantearse como una formación simple de registros. En otros contextos se le da también el nombre de *diccionario* o *relación*. Los esquemas de tabla son el fundamento de las *bases de datos relacionales*, aunque su implementación es muy diferente de las estructuras simplificadas que se presentan aquí.

Por ejemplo, podemos definir una tabla destinada a contener la identificación de las provincias:

```
typedef enum TipoProvincia {
  SinProvincia, Alava, Albacete,
  ... Zaragoza
};
const int MaxProvincias = int(Zaragoza);
typedef char TipoSiglas[2];
typedef char TipoNombre[30];

typedef struct TipoDatosProvincia {
  TipoSiglas siglas;
  TipoNombre nombre;
  int codigo;
};
typedef TipoDatosProvincia TipoTablaProvincias[MaxProvincias+1];

TipoTablaProvincias provincias;
```

En esta tabla se podrán almacenar los valores apropiados, asignándolos a los correspondientes campos, Por ejemplo:

```
provincias[Cadiz].siglas[0] = 'C';
provincias[Cadiz].siglas[1] = 'A';
strcpy( provincias[Cadiz].nombre, "Cádiz");
provincias[Cadiz].codigo = 11;
```

Estos datos podrán ser usados luego en programas que manejen direcciones postales, matrículas de coches, etc.

Pueden construirse estructuras de datos bastante complejas combinándolas de manera que en algunas de ellas hagamos referencia a datos almacenados en otras. Una forma de hacer referencia a los datos de una tabla es usando el índice correspondiente a la posición de cada registro. A continuación se muestra un ejemplo para manejo de figuras geométricas definidas a partir de puntos:

```
typedef struct TipoPunto {
  float x, y;
};
const int MaxPuntos = 1000;
typedef int TipoIndicePunto;
typedef TipoPunto TipoTablaPuntos[MaxPuntos];

typedef TipoIndicePunto TipoTriangulo[3];

typedef struct TipoCirculo {
  TipoIndicePunto centro;
  float radio;
};

TipoTablaPuntos puntos;
```

En este ejemplo, la tabla `puntos` almacena las coordenadas de todos los puntos utilizados para definir cualquiera de las figuras geométricas que se manejan. Las figuras se definen almacenando referencias a los puntos de la tabla, en lugar de almacenar directamente las coordenadas de los puntos. Un triángulo T definido por tres puntos A, B y C se podría registrar, por ejemplo, almacenando los puntos, arbitrariamente, en las posiciones 10, 11 y 12 de la tabla de puntos:

```
TipoTriangulo trianguloT;

LeerPunto( puntos[10] ); /* punto A */
LeerPunto( puntos[11] ); /* punto B */
LeerPunto( puntos[12] ); /* punto C */

trianguloT[1] = 10; /* punto A */
trianguloT[2] = 11; /* punto B */
trianguloT[3] = 12; /* punto C */
```

Como se decía, la tabla de puntos combina la información de todos los puntos de todas las figuras. Por ejemplo, se pueden definir círculos centrados en los vértices del triángulo T:

```
TipoCirculo c1, c2, c3;

c1.centro = 10; /* punto A */
c1.radio = 3.3;
c2.centro = 11; /* punto B */
c2.radio = 3.3;
c3.centro = 12; /* punto C */
c3.radio = 3.3;
```

El centro del círculo 1 es exactamente el mismo punto que el vértice A del triángulo. Si desplazamos ese punto se moverán a su vez el vértice y el círculo:

```
puntos[10].x = puntos[10].x + 2.5;
puntos[10].y = puntos[10].y + 4.3;
```

### 11.5.3 Ejemplo: Gestión de tarjetas de embarque

Este programa es una nueva versión más completa del ya presentado en el tema 10 para realizar la gestión de las plazas de un avión e imprimir las tarjetas de embarque. Ahora se puede asignar una plaza del avión de manera individual. También se puede reservar un grupo de dos a seis plazas de forma automática asignando asientos de la misma fila cuando sea posible. Antes de que las plazas asignadas automáticamente pasen a estar ocupadas se solicita confirmación de la operación.

Las plazas se gestionan como una tabla o vector de filas. Cada fila es un registro con un vector con el estado de cada asiento, e información precalculada del número y posición de las plazas contiguas que haya, con o sin pasillo en medio. Cada plaza puede estar vacía, ocupada o reservada. El estado de reservada es transitorio hasta que se confirma la asignación automática y pasa a estar ocupada, o no se confirma y vuelve a quedar libre. También se mantiene actualizado un contador del número total de plazas libres.

El procedimiento `PintarPlazas` es el encargado de mostrar la situación de todas las plazas del avión. El procedimiento `ImprimirTarjeta` es el encargado de escribir la tarjeta de embarque de una plaza del avión.

La función `BuscarPlazas` es la encargada de realizar el algoritmo para situar las plazas solicitadas. En un primer intento se trata de asignar las nuevas plazas en la misma fila y lado del pasillo. Como segundo intento se asignarán

plazas correlativas en la misma fila aunque tengan el pasillo en medio. Si eso tampoco es posible se irán asignando los asientos de uno en uno según haya sitios libres empezando por la primera fila. Cuando no hay suficientes plazas disponibles se dará un mensaje. Esta función utiliza las funciones **MarcarPlazasJuntas**, **MarcaPlazasJuntasConPasillo** y **MarcarPlazas** que implementan respectivamente cada una de las preferencias indicadas.

El procedimiento **ConfirmarPlazas** es el encargado de pasar las plazas reservadas bien a ocupadas o bien a vacías dependiendo de si se confirman o no las reservas. Además, este último procedimiento es el encargado de actualizar el número total de plazas libres después de la asignación.

El programa consiste en un menú con cuatro opciones (Plaza, Grupo, Estado y Fin). La opción de *Plaza* permite elegir y ocupar una plaza concreta si no está ocupada. La opción *Grupo* utiliza la función **BuscarPlazas** para la búsqueda de 2 a 6 plazas libres y reservarlas. Estas plazas deben ser confirmadas con el procedimiento **ConfirmarPlazas**. Para mostrar la situación de la ocupación del avión se dispone de la opción *Estado*.

Todo el programa está realizado en base a las constantes del número de filas NumFilas, y los asientos por fila AsientosFila. Cualquier tamaño de avión con el mismo formato se puede gestionar cambiando estas dos constantes. El listado completo del programa es el siguiente:

```
/*****************************************************
* Programa: Mostrador2
*
* Descripción:
*   Este programa confecciona tarjetas de embarque
*   y asigna plazas en la misma fila si es posible
******************************************************/
#include <stdio.h>
#include <ctype.h>

const int NumFilas = 10;     /* número de filas */
const int AsientosFila = 6; /* asientos por fila */
const int Pasillo = AsientosFila / 2;

typedef enum TipoEstado {vacio, ocupado, reservado};
typedef TipoEstado TipoOcupa[AsientosFila];
typedef struct TipoFila {
  TipoOcupa asientosOcupados;
  int juntas;  /* plazas seguidas al mismo lado */
  int desde;   /* desde ésta */
  int juntasP; /* plazas seguidas con pasillo en medio */
```

```
  int desdeP;  /* desde ésta */
};
typedef TipoFila TipoPlazas[NumFilas];

/** Mostrar ocupación del avión */
void PintarPlazas( const TipoPlazas P ) {
  const char DibujoAsiento[3] = { ' ', '*', 'R' };

  printf("\n      A    B    C       D    E    F\n\n");
  for (int i = 0; i < NumFilas; i++) {
    printf( "%3d", i + 1 );
    for (int j = 0; j < AsientosFila; j++) {
      if (j == Pasillo) {
        printf("   ");
      }
      printf( "  (%c)", DibujoAsiento[P[i].asientosOcupados[j]] );
    }
    printf( "\n" );
  }
  printf("\n");
}

/** Imprimir una "tarjeta de embarque" */
void ImprimirTarjeta( int fila, int asiento ) {
  printf( ".--------------------------.\n" );
  printf( "|    TARJETA DE EMBARQUE    |\n" );
  printf( "|  Fila :%3d", fila );
  printf( "  Asiento :%3c  |\n", asiento );
  printf( "'--------------------------'\n" );
}

/** Calcular número y posición de plazas contiguas */
void CalcularPlazasJuntas( TipoFila & fila ) {
  int juntas = 0;
  int juntasP = 0;

  fila.juntas = 0;
  fila.juntasP = 0;
  for (int j = 0; j < AsientosFila; j++) {
    if (j == Pasillo) {
      juntas = 0;
    }
    if (fila.asientosOcupados[j] == vacio) {
      juntas++;
      juntasP++;
      if (juntas > fila.juntas) {
```

```
        fila.juntas = juntas;
        fila.desde = j - juntas + 1;
      }
      if (juntasP > fila.juntasP) {
        fila.juntasP = juntasP;
        fila.desdeP = j - juntasP + 1;
      }
    } else {
      juntas = 0;
      juntasP = 0;
    }
  }
}

/** Reservar plazas contiguas al mismo lado de la fila */
bool MarcarPlazasJuntas( int numero, TipoFila & fila ) {
  if (fila.juntas < numero) {
    return false;
  } else {
    for (int j = 1; j <= numero; j++) {
      fila.asientosOcupados[fila.desde+j-1] = reservado;
    }
    return true;
  }
}

/** Reservar plazas contiguas a uno u otro lado de la fila */
bool MarcarPlazasJuntasConPasillo( int numero, TipoFila & fila ) {
  if (fila.juntasP < numero) {
    return false;
  } else {
    for (int j = 1; j <= numero; j++) {
      fila.asientosOcupados[fila.desdeP+j-1] = reservado;
    }
    return true;
  }
}

/** Reservar plazas contiguas o no en la fila */
void MarcarPlazas( int & numero, TipoFila & fila ) {
  int j = 0;

  while ( numero > 0 && j < AsientosFila ) {
    fila.asientosOcupados[j] = reservado;
    numero--;
  }
```

```
}

/** Buscar plazas contiguas o próximas */
bool BuscarPlazas( int nuevas, int libres, TipoPlazas plazas ) {
  int fila;

  if (nuevas <= libres) {
    /*-- Buscar plazas juntas en la misma fila --*/
    fila = 0;
    while (fila < NumFilas) {
      if (MarcarPlazasJuntas( nuevas, plazas[fila]) ) {
        return true;
      }
      fila++;
    }
    /*-- Buscar plazas seguidas en la misma fila --*/
    fila = 0;
    while (fila < NumFilas) {
      if (MarcarPlazasJuntasConPasillo(nuevas, plazas[fila])) {
        return true;
      }
      fila++;
    }
    /*-- Ocupar plazas de una en una desde la primera fila --*/
    fila = 0;
    while (nuevas > 0) {
      MarcarPlazas( nuevas, plazas[fila] );
    }
    return true;
  } else {
    printf("No hay plazas suficientes\n");
  }
  return false;
}

/** Confirmar o liberar plazas reservadas */
void ConfirmarPlazas( bool ok, TipoPlazas plazas ) {
  bool cambio;

  for (int i = 0; i < NumFilas; i++) {
    cambio = false;
    for (int j = 0; j < AsientosFila; j++) {
      if (plazas[i].asientosOcupados[j] == reservado) {
        if (ok) {
          plazas[i].asientosOcupados[j] = ocupado;
          ImprimirTarjeta(i + 1, char(int('A') + j));
```

```
          cambio = true;
        } else {
          plazas[i].asientosOcupados[j] = vacio;
        }
      }
    }
    if (cambio) {
      CalcularPlazasJuntas( plazas[i] );
    }
  }
}

/** Programa principal */
int main() {
  TipoPlazas pasaje;
  int sitiosLibres;
  char car;
  int aux, fil, col;
  bool seguir;

  /*-- Iniciar todo vacío --*/
  sitiosLibres = NumFilas * AsientosFila;
  for (int i = 0; i < NumFilas; i++) {
    for (int j = 0; j < AsientosFila; j++) {
      pasaje[i].asientosOcupados[j] = vacio;
    }
    CalcularPlazasJuntas( pasaje[i] );
  }

  /*-- Bucle de operaciones --*/
  seguir = true;
  while (seguir) {
    printf( "¿Opción (Plaza, Grupo, Estado, Fin)? ");
    scanf( " %c", &car );
    car = toupper(car);
    switch (car) {

        /*-- Asignar una plaza determinada --*/
      case 'P':
        do {
          printf( "¿Fila (1 a 10)? " );
          scanf( "%d", &aux );
        } while (aux < 1 || aux > 10);
        do {
          printf( "¿Asiento (A a F)? " );
          scanf( " %c", &car );
```

```
          car = toupper(car);
        } while (car < 'A' || car > 'F');
        fil = aux - 1;
        col = int(car - 'A');
        if (pasaje[fil].asientosOcupados[col] == vacio) {
          pasaje[fil].asientosOcupados[col] = ocupado;
          sitiosLibres--;
          CalcularPlazasJuntas( pasaje[fil] );
          ImprimirTarjeta(fil + 1, char(int('A') + col));
        } else {
          printf("** Plaza OCUPADA\n");
        }
        break;

        /*-- Asignar automáticamente plazas contiguas --*/
      case 'G' :
        do {
          printf( "¿Número de plazas (2 a 6)? ");
          scanf( "%d", &aux );
        } while (aux < 2 || aux > 6);
        if (BuscarPlazas( aux, sitiosLibres, pasaje )) {
          PintarPlazas( pasaje );
          printf( "¿Confirmar (S/N)? ");
          scanf( " %c", &car );
          if (toupper(car) == 'S') {
            ConfirmarPlazas( true, pasaje );
            sitiosLibres = sitiosLibres - aux;
          } else {
            ConfirmarPlazas( false, pasaje );
          }
        }
        break;

        /*-- Dibujar estado de ocupación --*/
      case 'E' :
        PintarPlazas( pasaje );
        break;

        /*-- Fin del programa --*/
      case 'F' :
        seguir = false;
        break;
    }
  }
}
```

A continuación se muestra un fragmento de la ejecución del programa:

```
...
¿Opción (Plaza, Grupo, Estado, Fin)? p
¿Fila (1 a 10)? 4
¿Asiento (A a F)? c
.--------------------------.
|    TARJETA DE EMBARQUE   |
|  Fila :  4  Asiento :  C |
'--------------------------'
¿Opción (Plaza, Grupo, Estado, Fin)? e

      A    B    C      D    E    F

  1  (*)  (*)  (*)    (*)  (*)  (*)
  2  (*)  (*)  (*)    (*)  (*)  (*)
  3  (*)  (*)  (*)    (*)  (*)  (*)
  4  ( )  ( )  (*)    ( )  ( )  ( )
  5  ( )  ( )  ( )    ( )  ( )  ( )
  6  ( )  ( )  ( )    ( )  ( )  ( )
  7  ( )  ( )  ( )    ( )  ( )  ( )
  8  ( )  ( )  ( )    ( )  ( )  ( )
  9  ( )  ( )  ( )    ( )  ( )  ( )
 10  ( )  ( )  ( )    ( )  ( )  ( )

¿Opción (Plaza, Grupo, Estado, Fin)? g
¿Número de plazas (2 a 6)? 4

      A    B    C      D    E    F

  1  (*)  (*)  (*)    (*)  (*)  (*)
  2  (*)  (*)  (*)    (*)  (*)  (*)
  3  (*)  (*)  (*)    (*)  (*)  (*)
  4  ( )  ( )  (*)    ( )  ( )  ( )
  5  (R)  (R)  (R)    (R)  ( )  ( )
  6  ( )  ( )  ( )    ( )  ( )  ( )
  7  ( )  ( )  ( )    ( )  ( )  ( )
  8  ( )  ( )  ( )    ( )  ( )  ( )
  9  ( )  ( )  ( )    ( )  ( )  ( )
 10  ( )  ( )  ( )    ( )  ( )  ( )

¿Confirmar (S/N)? s
.--------------------------.
|    TARJETA DE EMBARQUE   |
|  Fila :  5  Asiento :  A |
'--------------------------'
```

```
...
.--------------------------.
|    TARJETA DE EMBARQUE    |
|  Fila :  5  Asiento :  D  |
'--------------------------'
¿Opción (Plaza, Grupo, Estado, Fin)? g
¿Número de plazas (2 a 6)? 5

      A    B    C       D    E    F

  1  (*)  (*)  (*)     (*)  (*)  (*)
  2  (*)  (*)  (*)     (*)  (*)  (*)
  3  (*)  (*)  (*)     (*)  (*)  (*)
  4  ( )  ( )  (*)     ( )  ( )  ( )
  5  (*)  (*)  (*)     (*)  ( )  ( )
  6  (R)  (R)  (R)     (R)  (R)  ( )
  7  ( )  ( )  ( )     ( )  ( )  ( )
  8  ( )  ( )  ( )     ( )  ( )  ( )
  9  ( )  ( )  ( )     ( )  ( )  ( )
 10  ( )  ( )  ( )     ( )  ( )  ( )

¿Confirmar (S/N)? s
.--------------------------.
|    TARJETA DE EMBARQUE    |
|  Fila :  6  Asiento :  A  |
'--------------------------'
...
.--------------------------.
|    TARJETA DE EMBARQUE    |
|  Fila :  6  Asiento :  E  |
'--------------------------'
¿Opción (Plaza, Grupo, Estado, Fin)? g
¿Número de plazas (2 a 6)? 3

      A    B    C       D    E    F

  1  (*)  (*)  (*)     (*)  (*)  (*)
  2  (*)  (*)  (*)     (*)  (*)  (*)
  3  (*)  (*)  (*)     (*)  (*)  (*)
  4  ( )  ( )  (*)     (R)  (R)  (R)
  5  (*)  (*)  (*)     (*)  ( )  ( )
  6  (*)  (*)  (*)     (*)  (*)  ( )
  7  ( )  ( )  ( )     ( )  ( )  ( )
  8  ( )  ( )  ( )     ( )  ( )  ( )
  9  ( )  ( )  ( )     ( )  ( )  ( )
 10  ( )  ( )  ( )     ( )  ( )  ( )
```

```
¿Confirmar (S/N)? s
.---------------------------.
|    TARJETA DE EMBARQUE    |
|  Fila :  4  Asiento :  D  |
'---------------------------'
.---------------------------.
|    TARJETA DE EMBARQUE    |
|  Fila :  4  Asiento :  E  |
'---------------------------'
.---------------------------.
|    TARJETA DE EMBARQUE    |
|  Fila :  4  Asiento :  F  |
'---------------------------'
¿Opción (Plaza, Grupo, Estado, Fin)? f
```

# Tema 12

# Esquemas típicos de operación con formaciones

Cuando se trabaja con colecciones de datos es habitual que las operaciones globales sobre la colección se realicen a base de operar con los elementos uno a uno. Esto exige el empleo de esquemas de programas en los que se recorren los elementos de la colección en un orden adecuado.

De momento se han introducido las estructuras de tipo formación (vectores y matrices) para almacenar colecciones de datos. En este tema se introducen los *esquemas típicos de operación* con formaciones, incluyendo recorrido, búsqueda, inserción y ordenación así como ciertas técnicas particulares que facilitan su programación tales como *centinelas* y *matrices orladas*. Además, para los esquemas básicos se realiza previamente su *especificación formal* y se razona sobre la *corrección* de las implementaciones empleando los conceptos de *precondición*, *postcondición*, *invariante* y *variante* ya utilizados en los temas 6 y 8.

Al igual que entonces, se emplea una notación lógico-matemática convencional, con algunas simplificaciones y adaptaciones para hacerla más asequible a los usuarios de lenguajes de programación C, C++ y similares. La notación se describe de manera más precisa en el apéndice C.

En cualquier caso conviene recordar que las expresones lógico-matemáticas no triviales son relativamente difíciles de leer, por lo que siempre deberían ir acompañadas de una descripción en lenguaje natural que aclare el significado esperado.

## 12.1 Esquema de recorrido

El *esquema de recorrido* consiste en realizar cierta operación con todos y cada uno de los elementos de una formación (en algunos casos con parte de ellos). Evidentemente el esquema de recorrido se puede aplicar a formaciones de cualquier dimensión tales como matrices con dos o más índices. Sin embargo, para facilitar la compresión las explicaciones se circunscriben al caso de un vector (una dimensión).

La forma más general del esquema de recorrido sería:

```
iniciar operación
while (quedan elementos sin tratar) {
  elegir uno de ellos y tratarlo
}
completar operación
```

La corrección de este esquema, en términos generales, está garantizada por su propia construcción. Al final del bucle la condición de repetición (*quedan elementos sin tratar*) habrá dejado de cumplirse, y por tanto se cumplirá su complemento (*todos los elementos han sido tratados*). Por otra parte la terminación del bucle `while` está garantizada, ya que el número de elementos que faltan por tratar es un valor finito no negativo, que va disminuyendo en cada iteración, es decir, la expresión *variante* es simplemente el (*número de elementos sin tratar*).

Si es aceptable tratar todos los elementos en el orden de sus índices, el esquema de recorrido se puede concretar como un bucle `for`, más sencillo de entender. Para el caso de un vector `v` (una dimensión) con un número `N` de elementos (los índices van de `0` a `N-1`):

```
const int N = ...;
typedef ... T_Elemento ...;
typedef T_Elemento T_Vector[N];
T_Vector v;
...
iniciar operación
for (int i=0; i<N; i++) {
  tratar v[i]
}
completar operación
```

Como ejemplo, la siguiente función calcula la suma de los elementos de un vector abierto de números reales. Para razonar su corrección, el código se ha anotado con las aserciones correspondientes a la *precondición*, *postcondición*

y un *invariante* adecuado para el bucle. No es necesario identificar explícitamente la *variante* del bucle, ya que todo bucle **for** de **C±** tiene garantizada su terminación.

```
float SumaV( const float v[], int N ) {
«PRE : »
  float suma = 0;

  «INVARIANTE : suma = ∑v[0 .. i−1]»
  for (int i=0; i<N; i++) {
    suma = suma + v[i];
  }

  return suma;
«POST : SumaV(v,N) = ∑v[0 .. N−1]»
}
```

En este ejemplo la precondición queda vacía (vale siempre "*cierto*"), porque siempre se puede obtener la suma de los elementos del vector, incluso aunque tenga tamaño 0. La inicialización de la operación global consiste en poner a cero la variable `suma`. El invariante indica que la variable `suma` al comienzo de cada iteración contiene la suma de los elementos ya procesados. La terminación de la operación no exige ninguna acción adicional. La postcondición expresa que la variable `suma` al terminar la operación contiene la suma de todos los elementos del vector. Esto se deduce del hecho de que al final del recorrido todos los elementos han sido procesados.

Otro ejemplo ilustrativo es obtener el valor máximo de los números almacenados en un vector. En este caso el esquema es algo más complejo porque el máximo sólo existe si hay al menos un elemento, y además el tratamiento del primer elemento es diferente al de los demás. Si tratamos de seguir directamente el esquema inicial tendríamos lo siguiente:

```
float MaximoV( const float v[], int N ) {
«PRE : N > 0»
  float max;

  «INVARIANTE : i = 0 ∨ max = maximo(v[0 .. i−1])»
  for (int i=0; i<N; i++) {
    if (i==0 || v[i]>max) {
      max = v[i];
    }
  }
  return max;
«POST : MaximoV(v,N) = maximo(v[0 .. N−1])»
}
```

El invariante contiene una doble condición. En general nos dice que la variable **max** contiene en todo momento el valor máximo de los elementos que ya han sido procesados ($max = maximo(v[0..i-1])$), y por tanto al final tendrá el valor máximo de todos los elementos del vector. Como caso especial al comienzo del bucle ($i = 0$) no se impone ninguna restricción al valor de la variable **max**, ya que el máximo no está definido para una colección de cero elementos.

Si queremos evitar la doble condición **(i==0 || v[i]>max)** dentro del bucle, podríamos tratar el primer elemento por separado, fuera del bucle, y recorrer iterativamente los restantes elementos a partir del segundo:

```
float MaximoV( const float v[], int N ) {
«PRE : N > 0»
  float max;

  max = v[0];
  «INVARIANTE : max = maximo(v[0 .. i - 1])»
  for (int i=1; i<N; i++) {
    if (v[i]>max) {
      max = v[i];
    }
  }
  return max;
«POST : MaximoV(v, N) = maximo(v[0 .. N - 1])»
}
```

## 12.1.1 Recorrido de matrices

Si la formación es de tipo matriz, para hacer el *recorrido* se necesitan tantos **for** anidados como dimensiones tenga la formación. Por ejemplo, el recorrido es la operación típica que se debe realizar con una formación cuando se quieren inicializar todos sus elementos. El siguiente fragmento de código muestra cómo inicializar a cero todos los elementos de la una matriz **z** de números enteros. Se prescinde de formalizar los razonamientos de corrección por ser trivial en este caso, ya que el tratamiento de cada elemento es completamente independiente del de los demás, y no hay variables intermedias que vayan acumulando el resultado de cada repetición.

```
const int N = ...;
typedef int T_Matriz[N][N];
T_Matriz z;
...
```

```
for (int i=0; i<N; i++) {
  for (int j=0; j<N; j++) {
    z[i][j] = 0;
  }
}
```

Si se quieren escribir los valores de la matriz `z`, por filas, se puede utilizar el mismo esquema, en el que al finalizar cada fila se salta a una nueva línea:

```
printf( "\n" );
for (int i=0; i<N; i++) {
  for (int j=0; j<N; j++) {
    printf( "%5d", z[i][j] );
  }
  printf( "\n" );
}
```

La multiplicación de dos matrices `x` e `y` para dejar el resultado en la matriz `z` es un ejemplo muy interesante de recorrido. Primero se inicializa cada elemento de la matriz `z[i][j]` a cero utilizando el esquema anterior. A continuación, al mismo elemento inicializado se le van sumando los productos de cada uno de los elementos de la fila `i` de la matriz `x` por los correspondientes elementos de la columna `j` de la matriz `y` . Este cálculo de los productos para cada elemento se consigue mediante otro recorrido anidado al de inicialización, sobre los elementos de la fila/columna afectados:

```
for (int i=0; i<N; i++) {
  for (int j=0; j<N; j++) {
    z[i][j] = 0;
    for (int k=0; k<N; k++) {
      z[i][j] = z[i][j] + x[i][k]*y[k][j];
    }
  }
}
```

El razonamiento de corrección de los bucles externos es trivial. El razonamiento del bucle interno es similar al del ejemplo anterior de sumar los elementos de un vector.

### 12.1.2 Recorrido no lineal

En los ejemplos anteriores el índice usado para controlar el bucle nos señala directamente al elemento a procesar en cada iteración. En ciertos casos el elemento a procesar debe elegirse realizando ciertos cálculos, y el contador de

iteraciones sirve fundamentalmente para contabilizar el avance del recorrido y detectar el final del bucle.

Un ejemplo puede ser la construcción de un cuadrado mágico, en el que la suma de los números de cada fila, columna y diagonal principal es siempre la misma. Si el lado es impar, se puede construir rellenando las casillas con números correlativos, empezando por el centro de la fila superior y avanzando en diagonal hacia arriba y a la derecha. Al salir del cuadro por un lado se pasa a la casilla correspondiente del lado contrario. Si la siguiente casilla en avance diagonal ya está ocupada no se avanza, sino que se desciende a la casilla inmediatamente debajo para continuar el recorrido. Para un cuadrado de lado 3 el proceso sería el mostrado en la figura 12.1.

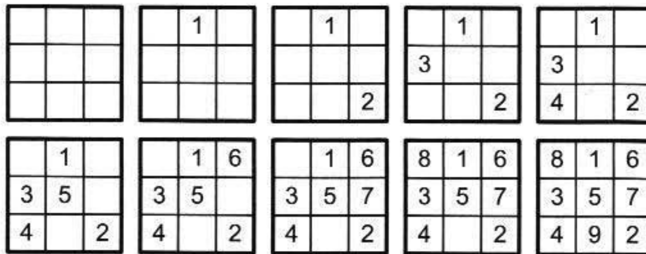


Figura 12.1 Recorrido no lineal: construcción de un cuadrado mágico.

El fragmento de código que rellena el cuadro de tamaño N, suponiendo que previamente todas las casilla tienen valor cero, sería:

```
const int N = ...;
typedef int T_Matriz[N][N];
T_Matriz cuadro;
int fil, col;
...
fil=0;
col=N/2;
for (int k = 1; k <= N*N; k++) {
  cuadro[fil][col] = k;
  fil = (fil+N-1) % N;
  col = (col+1) % N;
  if (cuadro[fil][col] != 0) {
    fil = (fil+2) % N;
    col = (col+N-1) % N;
  }
}
```

## 12.2 Búsqueda secuencial

En las operaciones de *búsqueda secuencial* se examinan uno a uno los elementos de la colección para tratar de localizar los que cumplen una cierta condición. Si realmente queremos encontrar todos los que existan, entonces la búsqueda equivale a un recorrido como los mostrados en la sección anterior. Como ejemplo, si queremos determinar el número de **Apariciones** de un cierto elemento **buscado** dentro de un vector **v** podremos utilizar el esquema de recorrido siguiente:

```
typedef ... T_Elemento ...;

int Apariciones( T_Elemento buscado, const T_Elemento v[], int N ) {
«PRE : »
  int veces = 0;

  «INVARIANTE : veces = cardinal(v[k = 0 .. i − 1], v[k] = buscado)»
  for (int i=0; i<N; i++) {
    if (v[i] == buscado) {
      veces++;
    }
  }
  return veces;
«POST : veces = cardinal(v[k = 0 .. N − 1], v[k] = buscado)»
}
```

Si no necesitamos localizar todos los elementos que cumplen la condición sino sólo uno de ellos, si lo hay, entonces no necesitamos recorrer la colección en su totalidad. El recorrido se debe detener en cuanto se encuentre el elemento buscado, y por tanto sólo será un recorrido completo cuando no se encuentre el elemento buscado dentro de la colección. Este planteamiento del problema indica que no se puede utilizar directamente una sentencia **for** de **C±** para la operación de búsqueda.

```
iniciar operación
while (quedan elementos sin tratar y no se ha encontrado ninguno aceptable) {
  elegir uno de ellos y ver si es aceptable
}
completar operación
```

Como en el apartado anterior, utilizaremos para las explicaciones un vector **v** de una sola dimensión con un número **N** de elementos (los índices van de **0** a **N-1**). El objetivo es localizar si hay algún elemento con un valor dado, y en este caso indicar su posición. Si no hay ningún elemento igual al buscado se

devuelve una posición negativa como indicación de fallo de la búsqueda. El esquema de búsqueda secuencial se puede plantear como:

```
typedef ... T_Elemento ...;

int Indice( T_Elemento buscado, const T_Elemento v[], int N ) {
«PRE : »
  int pos = 0;

  «INVARIANTE : Ningún elemento v[0..pos − 1] es el buscado»
  «INVARIANTE : buscado ∉ v[0..pos − 1]»
  while (pos<N && v[pos]!=buscado) {
    pos++;
  }
  if (pos>=N) {
    pos = -1;
  }
  return pos;
«POST : pos ≥ 0 → v[pos] = buscado ∧ pos < 0 → buscado ∉ v[0..N − 1]»
}
```

Cuando la búsqueda es infructuosa la posición **pos** llega a tomar un valor igual a N, que está fuera del margen superior del vector. La condición del bucle ya no se cumple y el bucle finaliza, pero dicha condición incluye una referencia al elemento `v[pos]` que no existe. Afortunadamente, como se explicó en el tema 5, el operador && de **C±** se evalúa "en cortocircuito". Cuando la primera condición `(pos<N)` ya es falsa no se continúa evaluando ninguna condición posterior dado que el resultado será necesariamente falso. Si se hubiese escrito `(v[pos]!=buscado && pos<N)` sí se podría intentar acceder al elemento `v[N]` que no existe y provocar errores graves en el programa.

## 12.3 Inserción

El problema que se plantea aquí es *insertar* un nuevo elemento en una colección de elementos ordenados, manteniendo el orden de la colección. Se supone que los elementos están almacenados en un vector, ocupando las posiciones desde el principio, y que queda algo de espacio libre al final del vector (si el vector estuviese lleno no se podrían insertar nuevos elementos).

La operación se puede realizar de forma iterativa, examinando los elementos empezando por el final hasta encontrar uno que sea inferior o igual al que se quiere insertar. Los elementos mayores que el que se quiere insertar se van moviendo una posición hacia adelante, con lo que va quedando un hueco en

medio del vector. Al encontrar un elemento menor que el nuevo, se copia el nuevo elemento en el hueco que hay en ese momento. La figura 12.2 muestra un ejemplo de inserción en una colección de valores numéricos enteros ordenados de manera creciente. El nuevo valor 10 debe insertarse entre el 8 y el 11.

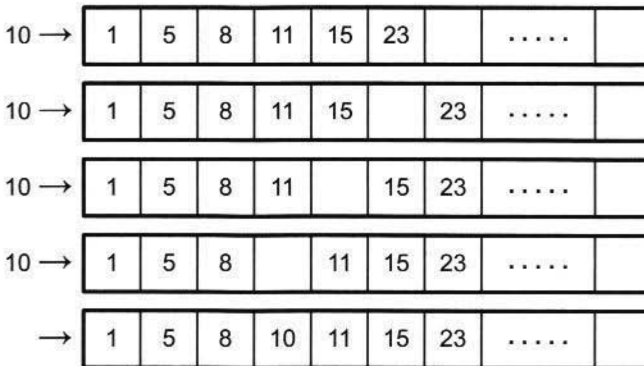



Figura 12.2 Inserción de un nuevo elemento.

Un esquema general del código de la operación sería:

```
Iniciar insercion
while ( ! Final && ! Encontrado hueco ) {
  Desplazar elemento
  Pasar al siguiente elemento
}
Insertar nuevo elemento
```

A continuación se concreta este código para la inserción de un nuevo `elemento` entre los `N` elementos de un `vector`, que ya están ordenados. Evidentemente, después de la inserción, el vector tendrá un elemento más. Por tanto, el valor de `N` siempre deberá ser menor que el tamaño del vector. En todo momento el índice auxiliar `j` señala al punto en el que hay hueco:

```
typedef ... T_Elemento ...;

void Insertar( T_Elemento v[], int N, T_Elemento elemento ) {
«PRE : v[0..N − 1] está ordenado»
  int j = N;

  «INVARIANTE : ...»
  while (j > 0 && elemento < v[j-1])  {
     v[j] = v[j-1];
     j--;
  }
  v[j] = elemento;
```

«*POST* : $v'[0..N]$ *está ordenado y*
*contiene los valores* $v[0..N-1]$ *más "elemento"*»
}

Es importante el orden en que se evalúan los términos de la condición del bucle. Si se hubiese escrito `(elemento < v[j-1] && j > 0)` sería posible intentar acceder al elemento `v[-1]`, que no existe.

En este ejemplo se ha omitido escribir el invariante en medio del código, y menos aún formalizarlo, por resultar muy prolijo. Dicho invariante, en lenguaje natural, es:

*«INVARIANTE: Los elementos delante del hueco están ordenados entre sí, y los elementos detrás del hueco están ordenados entre sí y son mayores que el nuevo, y los elementos delante del hueco son menores o iguales que los de detrás del hueco, y la colección de valores delante y detrás del hueco coincide con el contenido inicial del vector»*

Por su parte la expresión variante es obvia:

*«VARIANTE: Número de elementos delante del hueco»* ⇒ *«VARIANTE: j»*

## 12.4 Ordenación por inserción directa

En este apartado se aborda una solución para la ordenación de datos almacenados en un vector. Existen diversos métodos de ordenación de vectores cuyo estudio cae fuera del alcance de este libro. El método de *ordenación por inserción directa* es uno de los más sencillos y está basado en el esquema de inserción mostrado en el apartado anterior. Por ejemplo, se trata de ordenar un vector `v` de diez elementos (índices de 0 a 9) y que inicialmente está desordenado, tal como se muestra en la figura 12.3.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 21 | 5 | 3 | 12 | 65 | 9 | 36 | 7 | 2 | 45 |

Figura 12.3 Vector inicial.

Para comenzar, el primer elemento (`21`) está ya ordenado consigo mismo. A continuación, extraemos el segundo elemento (`5`) y se genera un hueco, que se puede utilizar para ampliar la parte del vector ya ordenada. El método de ordenación consiste en insertar el elemento extraído en su lugar correspondiente entre los elementos ya ordenados. Este proceso se repite con el tercero, cuarto, quinto, ... y décimo elemento hasta quedar ordenado todo el vector.

La secuencia de inserciones de los sucesivos elementos del vector se muestra en la figura 12.4.

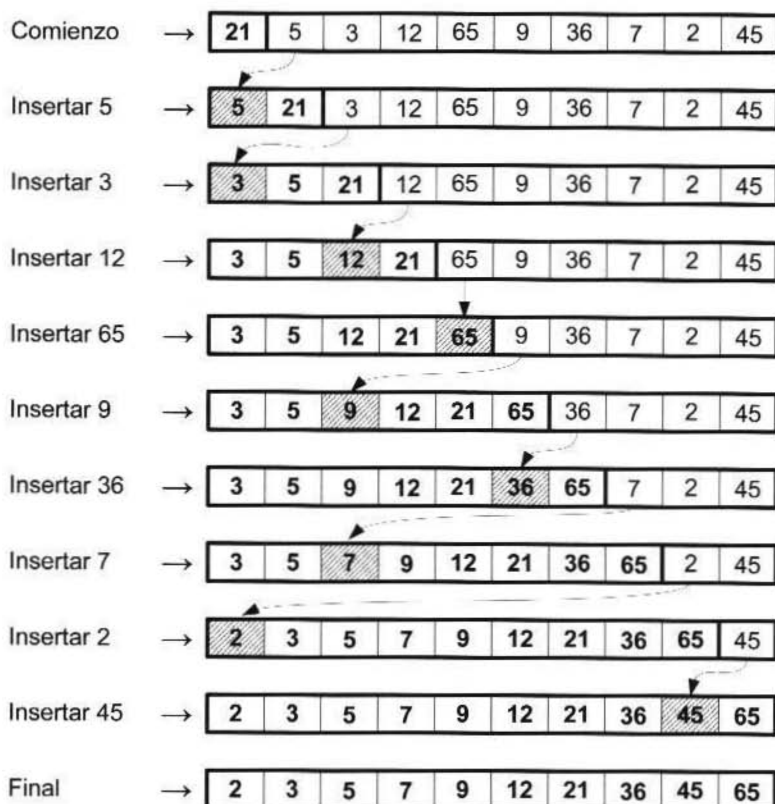



Figura 12.4 Secuencia de inserciones para la ordenación.

En la secuencia se muestra con distinto fondo la posición del hueco inmediatamente antes de la inserción del siguiente elemento. El final de la parte del vector ya ordenada se marca con una línea de mayor grosor. Además, los números ya ordenados también están en negrita. La inserción se realiza desde la posición anterior a la del elemento extraído.

A continuación se muestra el código de una posible realización. La variable `valor` guarda el elemento extraído de la posición `i`. La ordenación total del vector se consigue mediante el recorrido de todos sus elementos, desde el segundo, para buscarles su hueco e insertarlos en él.

```
typedef ... T_Elemento ...;
```

```
void Ordenar( T_Elemento v[], int N ) {
«PRE: »
  T_Elemento valor;
  int j;

  «INVARIANTE: v[0..i-1] está ordenado»
  «INVARIANTE: ∀k ∈ (1..i-1) • v[k] ≥ v[k-1]»
  for (int i=1; i<N; i++) {
    valor = v[i];
    j = i;
    while (j > 0 && valor < v[j-1]) {
      v[j] = v[j-1];
      j--;
    }
    v[j] = valor
  }
«POST: v'[0..N-1] está ordenado y contiene los valores v[0..N-1]»
}
```

## 12.5 Búsqueda por dicotomía

Cuando los datos están ordenados la búsqueda resulta mucho mas rápida. Si comparamos el elemento a buscar con el que está justo en la mitad de los datos ordenados, podemos decidir si es ése el elemento que buscamos o debemos continuar buscando, pero sólo en la mitad derecha o sólo en la mitad izquierda.

El mismo proceso se puede repetir con la mitad elegida, comparando con el elemento que está en el centro de dicha mitad. En cada comparación, la búsqueda se reduce a comprobar si el dato buscado está entre la mitad de los anteriores. La búsqueda finaliza cuando el elemento se encuentra, o bien cuando la zona pendiente de examinar queda reducida a un sólo elemento (o ninguno) después de sucesivas divisiones en mitades. Esta búsqueda se denomina *búsqueda por dicotomía*. El esquema general de búsqueda sigue siendo análogo al utilizado anteriormente. Sin embargo, ahora es necesario cambiar la obtención del siguiente elemento a comparar:

```
iniciar operación
while (quedan elementos por examinar y no se ha encontrado ninguno aceptable) {
  elegir el elemento central y ver si es aceptable
}
completar operación
```

Como ejemplo, realizaremos la búsqueda por dicotomía de un valor **buscado** en un vector `v` de `N` elementos. Se necesitan dos variables para acotar el trozo de vector que todavía queda por comprobar y otra variable para señalar el punto medio de ambas. Estas variables las denominaremos `izq` , `dch` y `mitad`. También usaremos una variable **pos** para almacenar finalmente el resultado de la búsqueda, es decir, el lugar en el que se ha encontrado el elemento buscado, o bien un valor negativo (p. ej. -1) si el elemento buscado no se encuentra. El código sería como sigue:

```
typedef ... T_Elemento ...;

int Indice( T_Elemento buscado, const T_Elemento v[], int N ) {
«PRE : v está ordenado»
  int izq, dch, mitad, pos;
  izq = 0; dch = N-1; pos = -1;

  «INVARIANTE : el elemento buscado, si existe, está en la zona v[izq..dch]»
  «INVARIANTE : ∀k ∈ (0..izq − 1) • (v[k] < buscado) ∧
                ∀k ∈ (dch + 1..N − 1) • (v[k] > buscado)»
  «VARIANTE : dch − izq + 1»
  while (pos < 0 && izq <= dch) {
    mitad = (izq + dch) / 2;
    if (v[mitad] == buscado) {
      pos = mitad;
    } else if (v[mitad] < buscado) {
      dch = mitad - 1;
    } else {
      izq = mitad + 1;
    }
  }
  return pos;
«POST : pos ≥ 0 → v[pos] = buscado  ∧  pos < 0 → buscado ∉ v[0..N − 1]»
}
```

Inicialmente se busca en el vector completo `v[izq..dch] = v[0..N-1]`. La variable auxiliar **pos** se inicializa al valor de fallo de búsqueda (-1), y permanece con dicho valor hasta que se encuentre el elemento buscado, si existe, en cuyo caso señala a la posición encontrada. En cada etapa se examina el elemento central de la zona de búsqueda, `mitad=(izq+dch)/2)`, y si `v[mitad]` no contiene el elemento buscado la zona `v[izq..dch]` se reduce a la mitad de delante o a la de detrás. El bucle termina cuando se encuentra el valor buscado o bien cuando la zona de búsqueda ya no contiene elementos (`izq>dch`).

La terminación se garantiza fácilmente. Como variante se puede utilizar el tamaño de la zona de búsqueda `dch-izq+1`, que se reduce estrictamente a la

mitad (cociente entero por defecto) en cada paso, y por tanto se reduce a cero en un número finito de pasos.

La búsqueda por dicotomía se puede aplicar, por ejemplo, en el esquema de ordenación por inserción, para localizar el lugar en que hay que insertar el nuevo elemento.

## 12.6 Simplificación de las condiciones de contorno

La programación de operaciones con vectores, realizadas elemento a elemento, exige con frecuencia realizar un tratamiento especial de los elementos extremos del vector o, en general, de los elementos del *contorno* de una formación. A continuación veremos algunas técnicas particulares para evitar la necesidad de detectar de manera explícita si se ha llegado a un elemento del contorno y/o realizar con él un tratamiento especial.

### 12.6.1 Técnica del centinela

Por ejemplo, en el procedimiento general de búsqueda es necesario comprobar en cada iteración una condición doble: si no se ha alcanzado todavía el final del vector, y si se encuentra el elemento buscado. Repetimos aquí el esquema de la sección 12.2:

```
iniciar operación
while (quedan elementos sin tratar y no se ha encontrado ninguno aceptable) {
  elegir uno de ellos y ver si es aceptable
}
completar operación
```

La doble condición del bucle de iteración complica el código y supone un tiempo adicional en la ejecución de cada iteración. Si garantizamos que el dato buscado está dentro de la zona de búsqueda, antes o después se terminará encontrándolo y ya no será necesario comprobar explícitamente si se alcanza el final del vector.

La manera de asegurar esto es incluir el dato buscado en el vector antes de comenzar la búsqueda. El vector se amplía, según se muestra en la figura 12.5, con un elemento más situado al final (si se hace la búsqueda hacia adelante) o situado al principio (si la búsqueda se hace hacia atrás). En ese elemento adicional se copia el dato a buscar antes de iniciar la búsqueda para que actúe como *centinela* (C) y asegure que la búsqueda nunca acaba de forma infructuosa.

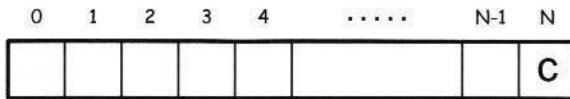


Figura 12.5 Vector con centinela situado al final.

Ahora, el esquema general de búsqueda se simplifica de la siguiente forma:

```
iniciar operación (colocar el centinela)
while (no se ha encontrado un elemento aceptable) {
  elegir otro elemento y ver si es aceptable
}
completar operación (si se ha encontrado el centinela, indicar fallo en la búsqueda)
```

Igual que en los ejemplos anteriores, usaremos una variable **pos** para almacenar finalmente el resultado de la búsqueda, es decir, el lugar en el que se ha encontrado el elemento buscado, o bien un valor negativo (p. ej. -1) si el elemento buscado no se encuentra. El fragmento de programa sería como sigue:

```
typedef ... T_Elemento ...;

int Indice( T_Elemento buscado, const T_Elemento v[], int N ) {
  int pos = 0;

  v[N] = buscado;  /* centinela */
  while (v[pos]!=buscado) {
    pos++;
  }
  if (pos>=N) { /* lo que se encuentra es el centinela */
    pos = -1;
  }
  return pos;
}
```

En este ejemplo la búsqueda se hace hacia adelante. El centinela se coloca al final, detrás del último elemento del vector original, es decir, en la posición N. Si al final de la búsqueda la variable **pos** tiene un valor igual a N, quiere decir que la búsqueda ha sido infructuosa: el elemento encontrado ha sido el centinela, y no un elemento del vector original.

La técnica del centinela se puede emplear en la ordenación por inserción, aprovechando que cada nuevo elemento a insertar entre los anteriores está ya situado al final de la parte ordenada. Ahora la localización del lugar que le corresponde y el movimiento de los elementos posteriores para hacer hueco se

hacen por separado, y la búsqueda entre los elementos anteriores se hace hacia adelante y no hacia atrás. La nueva redacción del programa es la siguiente:

```
typedef ... T_Elemento ...;

void Ordenar( T_Elemento v[], int N ) {
  T_Elemento valor;
  int j;

  for (int i=1; i<N; i++) {
    valor = v[i];
    /*-- Buscar la nueva posición del elemento, sin mover nada --*/
    j = 0;
    while (valor > v[j]) {
      j++;
    }
    /*-- Mover elementos mayores y poner el elemento en su sitio --*/
    for (int k=i-1; k>=j; k--) {
      v[k+1] = v[k];
    }
    v[j] = valor
  }
}
```

Como se puede observar, la técnica del centinela simplifica la condición del bucle para la operación de buscar la nueva posición del elemento.

### 12.6.2 Matrices orladas

Cuando se trabaja con matrices, también se pueden simplificar las condiciones de contorno utilizando *matrices orladas*. Estas matrices se dimensionan con dos filas y dos columnas más de las necesarias tal como se muestra en la figura 12.6.

La matriz original la forman las filas 1 a M y las columnas 1 a N. Las filas y columnas extra garantizan que todos los elementos de la matriz original tienen elementos vecinos en todas las direcciones. Antes de operar con la matriz se inicializan las filas y columnas extra con un valor de contorno (`C`) que permita simplificar la operación de modo que el tratamiento de los elementos del borde de la matriz original sea idéntico al de los demás.

Para ilustrar el uso de una matriz orlada supongamos que tenemos una imagen en blanco y negro (escala de grises) de `Ancho` × `Alto` pixeles almacenada en una matriz definida de la siguiente forma:

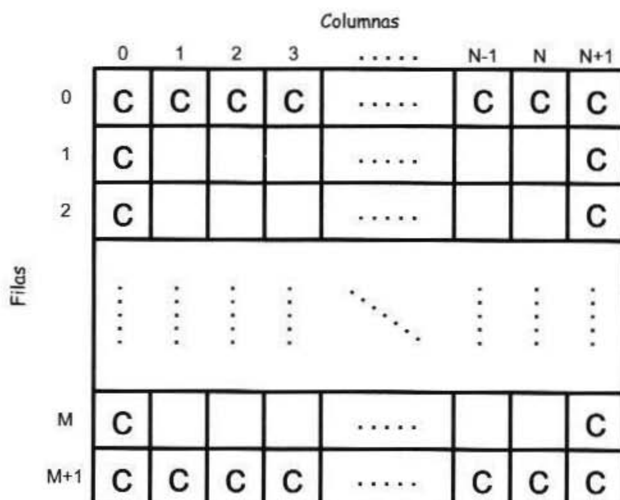



Figura 12.6 Matriz orlada.

```
const int Ancho  = 40;  /* Anchura de la imagen */
const int Alto   = 20;  /* Altura de la imagen */
const int Borde  = -1;  /* Indicador de borde de la imagen */
const int Blanco = 0;   /* Nivel bajo de grises = blanco */
const int Negro  = 5;   /* Nivel alto de grises = negro */

typedef int Imagen_t[Alto+2][Ancho+2];
Imagen_t imagen;
```

Los puntos de la imagen ocupan las columnas `1` a `Ancho` y las filas `1` a `Alto`. Cada elemento de la matriz guarda el nivel de gris del correspondiente punto de la imagen. Las columnas `0` y `Ancho+1` y las filas `0` y `Alto+1` son los elementos extra del contorno. Para tratar cada punto de la imagen individualmente basta hacer un recorrido completo, sin ninguna complicación. Por ejemplo, para contrastar la imagen y reducir todos los puntos gris claro a blanco y todos los gris oscuro a negro, bastará escribir:

```
for (int i=1; i<=Alto; i++) {
  for (int j=1; j<=Ancho; j++) {
    if (imagen[i][j] <= nivel) {
      imagen[i][j] = Blanco;
    } else {
      imagen[i][j] = Negro;
    }
  }
}
```

En el fragmento de código anterior no se ha usado para nada el contorno de la matriz. Supongamos ahora que queremos recortar la imagen, esto es, eliminar aquellos puntos externos de la imagen que están en blanco, pero dejando los puntos blancos que son interiores y están rodeados de puntos negros. El tratamiento de cada punto exige examinar al mismo tiempo los puntos contiguos. El programa se simplifica si se garantiza que todo punto útil de la imagen tiene puntos contiguos en todas las direcciones, es decir, no hay situaciones excepcionales en los bordes.

Para ello se aprovecha el contorno de la matriz, es decir, la *orla*, inicializándola con el valor que indicará los elementos del borde.

```
for (int i=0; i<=Alto+1; i++) {
  imagen[i][0] = Borde;
  imagen[i][Ancho+1] = Borde;
}
for (int i=0; i<=Ancho+1; i++) {
  imagen[0][i] = Borde;
  imagen[Alto+1][i] = Borde;
}
```

Suponiendo que `imagen` está ya contrastada anteriormente y que su contorno está inicializado al valor de `Borde`, el recorte se realizaría mediante sucesivos recorridos de toda la matriz en los que los puntos blancos que tienen algún punto `Borde` alrededor deben pasar también a puntos `Borde`. El proceso de recorte se termina cuando en un recorrido completo de toda la imagen no hay ningún punto que cambie. El fragmento de programa que realiza el recorte es el siguiente:

```
bool fin;

do {
  fin = true;
  for (int i=1;i<=Alto;i++) {
    for (int j=1;j<=Ancho;j++) {
      if ((imagen[i][j] == Blanco) &&
          ((imagen[i-1][j] == Borde) ||
           (imagen[i][j-1] == Borde) ||
           (imagen[i][j+1] == Borde) ||
           (imagen[i+1][j] == Borde))) {
        imagen[i][j] = Borde;
        fin = false;
      }
    }
  }
} while (! fin);
```

Esta forma de operar se denomina de *fuerza bruta*, y puede ser poco eficiente, pero es muy sencilla de programar.

Este ejemplo es un fragmento de uno de los programas completos que aparecen en este tema.

## 12.7 Ejemplos de programas

A continuación se muestran varios programas completos con ejemplos de sus respectivas ejecuciones.

### 12.7.1 Ejemplo: Sopa de letras

Con este programa se trata de realizar la búsqueda de una palabra dentro de una matriz de caracteres. La búsqueda debe realizarse en horizontal, vertical y diagonal en ambos sentidos. Por tanto, se pueden establecer 8 direcciones de búsqueda: norte, sur, este, oeste, noroeste, suroeste, noreste, sureste. En la búsqueda se deben tener en cuenta los límites de la matriz. La función `Buscar` comprueba si la palabra `palabra` de longitud `letras` coincide con los caracteres de la matriz desde la posición: `fila` y `columna`, siguiendo la dirección `rumbo`. Esta función tiene en cuenta los límites de la matriz y devuelve un resultado cierto o falso según se encuentre o no la palabra buscada.

Los demás procedimientos son auxiliares. El procedimiento `IniciarSopa` inicializa de forma aleatoria la matriz, mediante un recorrido de la misma. Cada carácter se obtiene a partir de un número aleatorio entre 0 y 26, al que se suma la posición del carácter `'a'` dentro de la tabla ASCII.

El procedimiento `Cambiar`, es el encargado de pasar de minúsculas a mayúsculas la palabra encontrada. Este cambio se realiza mediante un recorrido parcial de `letras` caracteres de la matriz desde la posición: `fila` y `columna`, siguiendo la dirección `rumbo`.

El procedimiento `EscribirSopa` es un recorrido total de la matriz en el que se escriben por filas todos los caracteres de la matriz separados por un blanco.

En el programa principal se lee la palabra a buscar, toda en minúsculas, e inmediatamente se realiza su búsqueda exhaustiva desde todas las posiciones de la matriz y con todas las direcciones posibles. Si se encuentra, se cambia a mayúsculas. El listado del programa es el siguiente:

```
/**************************************************
* Programa: SopaDeLetras
*
* Descripción:
*   Este programa busca una palabra en una matriz
*   de caracteres, en cualquier dirección, de forma
*   semejante a como se hace en una sopa de letras
**************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>

const int Filas = 10;        /* Filas de la sopa de letras */
const int Columnas = 20;     /* Columnas de la sopa de letras */
const int TotalLetras = 26; /* Total letras abecedario inglés */

typedef char TipoSopa[Filas][Columnas];
typedef char TipoPalabra[Columnas];
typedef enum TipoRumbo {Norte, Sur, Este, Oeste,
                        Noroeste, Suroeste, Noreste, Sureste
                       };

const int PasoH[8] = { 0, 0, 1, -1, -1, -1, 1, 1 };
const int PasoV[8] = { -1, 1, 0, 0, -1, 1, -1, 1 };

/*========================================
  Procedimiento para inicializar de manera
  aleatoria una sopa de letras
  ========================================*/
void IniciarSopa( TipoSopa sopa ) {
  srand( time(NULL) );
  for (int f=0; f<Filas; f++) {
    for (int c=0; c<Columnas; c++) {
      sopa[f][c] = char(rand()%TotalLetras+int('a'));
    }
  }
}

/*=======================================================
  Procedimiento para cambiar "letras" a mayúsculas
  desde la posición (columna,fila) con el "rumbo" indicado
  =======================================================*/
void Cambiar( TipoSopa sopa, int letras,
              int columna, int fila, TipoRumbo rumbo ) {
```

```
  for (int i=1; i<=letras; i++) {
    sopa[fila][columna] = toupper(sopa[fila][columna]);
    /*-- Nueva fila y columna según el rumbo --*/
    columna = columna + PasoH[rumbo];
    fila = fila + PasoV[rumbo];
  }
}

/*============================================================
  Función para buscar las "letras" de una "palabra",
  desde la posición (columna,fila), con el "rumbo" indicado,
  en la "sopa" de letras
  ============================================================*/
bool Buscar( const TipoSopa sopa, const TipoPalabra palabra, int letras,
             int columna, int fila, TipoRumbo rumbo ) {
  int i = 0; /* contador de letras */
  bool coincide = true; /* hay coincidencia */

  /*-- Invariante: i = nº de letras examinadas
                 y  coincide = "las i letras coinciden" --*/
  while (i < letras && coincide && fila >= 0 && fila < Filas &&
         columna >= 0 && columna < Columnas) {
    /*-- Se acepta coincidencia en mayúscula o minúscula --*/
    coincide = palabra[i] == tolower(sopa[fila][columna]);
    if (coincide) {
      i++;
      /*-- Nueva fila y columna según el rumbo --*/
      columna = columna + PasoH[rumbo];
      fila = fila + PasoV[rumbo];
    }
  }
  return i==letras;
}

/*============================================
  Procedimiento para escribir la sopa de letras
  ============================================*/
void EscribirSopa( const TipoSopa sopa ) {
  printf( "\n" );
  for (int i=0; i<Filas; i++) {
    for (int j=0; j<Columnas; j++) {
      printf( "%c ",sopa[i][j] );
    }
    printf( "\n" );
  }
}
```

```
/*==================
  Programa principal
  ==================*/
int main() {
  TipoSopa sopa;
  TipoPalabra palabra;
  TipoRumbo rumbo;
  int longitud;
  char tecla;  /* último caracter leído */
  int encontradas;

  /*-- Crear la sopa de letras al azar --*/
  IniciarSopa( sopa );
  EscribirSopa( sopa );

  /*-- Búsqueda de palabras --*/
  do {
    printf( "\n¿Palabra a Buscar? " );
    /*-- Leer palabra a buscar, saltando blancos iniciales --*/
    scanf( " %c", &tecla ); /* primera letra */
    longitud = 0;
    while (islower(tecla)) {
      palabra[longitud] = tecla;
      longitud++;
      scanf( "%c", &tecla );
    }

    /*-- Buscar desde todos los puntos posibles
         y en todas las direcciones posibles --*/
    encontradas = 0;
    for (int iFila = 0; iFila < Filas; iFila++) {
      for (int iColumna = 0; iColumna < Columnas; iColumna++) {
        for (int iRumbo = int(Norte); iRumbo <= int(Sureste); iRumbo++) {
          rumbo = TipoRumbo(iRumbo);
          if (Buscar( sopa, palabra, longitud, iColumna, iFila, rumbo)) {
            /*-- Cuando se encuentra se cambia a mayúsculas --*/
            Cambiar( sopa, longitud, iColumna, iFila, rumbo);
            encontradas++;
          }
        }
      }
    }

    /*-- Mostrar resultado --*/
    if (encontradas > 0) {
      printf( "\n%d coincidencias\n", encontradas );
```

```
      EscribirSopa( sopa );
    } else {
      printf( "\nNo encontrada\n" );
    }

    /*-- Repetir búsqueda --*/
    printf( "\n¿Otra Palabra(S/N)? " );
    do {
      scanf( " %c", &tecla );
      tecla = toupper(tecla);
    } while (tecla != 'S' && tecla != 'N');
  } while (tecla != 'N');
}
```

La ejecución del programa produce un resultado similar al siguiente:

```
t d w j q p f u e r w s y i o o p s p w
p a l j m s x e b f k v h x e e p w u n
f h i r w w l y z f q d b h n c y g r l
y v f a e q j x u h i o c a s z f k o r
j w j c c o y b r f t j z b e r a b j j
j v u o i c d o x d b r r v o y d v y t
y z i k n w s m k y x h v c b i e r i r
t g z n q p y s w r q z b i m e n m q n
o j a q b n g q z x q b d d z p x h u e
w l b d q a n a m a t p x t u i y h z p

¿Palabra a Buscar? pena

1 coincidencias

t d w j q p f u e r w s y i o o P s p w
p a l j m s x e b f k v h x e E p w u n
f h i r w w l y z f q d b h N c y g r l
y v f a e q j x u h i o c A s z f k o r
j w j c c o y b r f t j z b e r a b j j
j v u o i c d o x d b r r v o y d v y t
y z i k n w s m k y x h v c b i e r i r
t g z n q p y s w r q z b i m e n m q n
o j a q b n g q z x q b d d z p x h u e
w l b d q a n a m a t p x t u i y h z p

¿Otra Palabra(S/N)? s

¿Palabra a Buscar? dos
```

```
No encontrada

¿Otra Palabra(S/N)? s

¿Palabra a Buscar? voy

1 coincidencias

t d w j q p f u e r w s y i o o P s p w
p a l j m s x e b f k v h x e E p w u n
f h i r w w l y z f q d b h N c y g r l
y v f a e q j x u h i o c A s z f k o r
j w j c c o y b r f t j z b e r a b j j
j v u o i c d o x d b r r V O Y d v y t
y z i k n w s m k y x h v c b i e r i r
t g z n q p y s w r q z b i m e n m q n
o j a q b n g q z x q b d d z p x h u e
w l b d q a n a m a t p x t u i y h z p

¿Otra Palabra(S/N)? s

¿Palabra a Buscar? ir

3 coincidencias

t d w j q p f u e r w s y i o o P s p w
p a l j m s x e b f k v h x e E p w u n
f h I R w w l y z f q d b h N c y g r l
y v f a e q j x u h i o c A s z f k o r
j w j c c o y b r f t j z b e r a b j j
j v u o i c d o x d b r r V O Y d v y t
y z i k n w s m k y x h v c b i e R I R
t g z n q p y s w r q z b i m e n m q n
o j a q b n g q z x q b d d z p x h u e
w l b d q a n a m a t p x t u i y h z p

¿Otra Palabra(S/N)? n
```

### 12.7.2 Ejemplo: Imprimir fechas en orden

En este ejemplo se lee una colección de fechas, comprobando que son correctas, y a continuación se imprimen, ordenadas cronológicamente.

Las fechas se leen de la entrada principal. Cada fecha de entrada comprende tres valores, correspondientes al día, mes y año. El día y el año se dan en

forma numérica. El mes puede darse como número o en letra y por ello se utiliza un esquema **union** con dos campos alternativos: un número entero o bien un vector de hasta 15 caracteres.

El año puede darse completo, con cuatro dígitos, o en forma abreviada, con dos. En este último caso se entiende que es un año de nuestro siglo, y se le suma 2000. La lectura de datos termina al introducir un valor numérico cero para el día.

Las fechas leídas se comprueban para garantizar que el día, mes y año forman una combinación consistente. Las fechas erróneas se descartan.

Las fechas correctas se van almacenando en una tabla de fechas, que se mantiene ordenada en todo momento. Cada nueva fecha leída se inserta en la posición que le corresponde, en orden cronológico. Al final se imprime toda la colección de fechas, en el orden en que se han almacenado.

A continuación se presenta un ejemplo de ejecución del programa. Los datos de entrada son:

```
10 Marzo 1972
30 feb 82
29 FEB 1900
11 3 72
29 FEB 2000
28 diciem 1993
4 enero 91
15 error 89
10 10 10
0
```

El resultado obtenido a la salida es:

```
Fechas leídas:
10-Marzo-1972
30-Febrero-2082 ** incorrecta **
29-Febrero-1900 ** incorrecta **
11-Marzo-2072
29-Febrero-2000
28-Diciembre-1993
 4-Enero-2091
15- -2089 ** incorrecta **
10-Octubre-2010

Fechas en orden:
10-Marzo-1972
```

```
28-Diciembre-1993
29-Febrero-2000
10-Octubre-2010
11-Marzo-2072
 4-Enero-2091
```

El listado completo del programa es el siguiente:

```
/************************************************************
*    Programa: Fechas
*
* Descripción:
*   Programa que lee una serie de fechas, comprueba que son
*   correctas, y las imprime en orden cronológico.
************************************************************/
#include <stdio.h>
#include <ctype.h>

typedef struct fecha_t {
  int dia;
  int mes;
  int anno;
};

const int maxNombre = 15;
typedef char nombreMes_t[maxNombre];
typedef union dato_t {
  int mesNumero;
  nombreMes_t mesLetra;
};
typedef struct datoMes_t {
  bool esNumero;
  dato_t dato;
};

const int maxFechas = 100;
typedef fecha_t listaFechas_t[maxFechas];

listaFechas_t lista;          /* lista de fechas leídas */
int numFechas;                /* número de fechas leídas */

typedef nombreMes_t listaNombres_t[13];
listaNombres_t nombres  = {" ",         /* sin nombre cuando mes = 0 */
                           "Enero",   /* nombres de los meses */
                           "Febrero",
                           "Marzo",
```

```
                              "Abril",
                              "Mayo",
                              "Junio",
                              "Julio",
                              "Agosto",
                              "Septiembre",
                              "Octubre",
                              "Noviembre",
                              "Diciembre"
                            };

/*============================================
  Comprobar si una fecha es correcta
  ============================================*/
bool EsCorrecta( const fecha_t fecha ) {
  if ((fecha.anno<=0) || (fecha.dia<=0)) {
    return false;
  }
  switch (fecha.mes) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
      return (fecha.dia <= 31);
      break;
    case 4:
    case 6:
    case 9:
    case 11:
      return (fecha.dia <= 30);
      break;
    case 2:
      if (fecha.anno%4 == 0 && fecha.anno%100 != 0 ||
          fecha.anno%400 == 0) {
        return (fecha.dia <= 29);
      } else {
        return (fecha.dia <= 28);
      };
      break;
    default :
      return false;
  }
}
```

```
/*=====================================================
  Imprimir una fecha con el formato  dd-nombremes-aaaa
  =====================================================*/
void EscribirFecha ( const fecha_t fecha ) {
  printf( "%2d-%s-%4d", fecha.dia, nombres[fecha.mes], fecha.anno );
}

/*=============================
  Leer un mes en número o letra
  =============================*/
void LeerMes( datoMes_t & mes ) {
  char c;
  int k;

  do {
    scanf( " %c", &c );   /* leer primer carácter */
  } while (! isalnum(c)); /* ignorando puntuación */
  if (isdigit(c)) {       /* mes como número */
    mes.esNumero = true;
    mes.dato.mesNumero = 0;
    do {
      mes.dato.mesNumero = mes.dato.mesNumero*10 + int(c)-int('0');
      scanf( "%c", &c );
    } while (isdigit(c));
  } else {                /* mes en letra */
    mes.esNumero = false;
    k = 0;
    do {
      if (k<maxNombre) {
        mes.dato.mesLetra[k] = c;
        k++;
      }
      scanf( "%c", &c );
    } while (isalpha(c));
  }
}

/*=============================================
  Convertir a número el mes en letra. Sólo se
  comprueban los tres primeros caracteres.
  Si no es correcto se devuelve cero.
  =============================================*/
int NumeroDelMes( const nombreMes_t nombre ) {
  int k;
```

```
  for (int mes=1; mes<=12; mes++) {
    k = 0;
    while (toupper(nombre[k]) == toupper(nombres[mes][k])) {
      if (k >= 2) {  /* coinciden 3 caracteres */
        return mes;
      }
      k++;
    }
  }
  return 0;
}

/*=============================================
  Leer una fecha con el mes en número o letra.
  Si el día es cero, se asume fin de los datos
  =============================================*/
void LeerFecha( fecha_t & fecha ) {
  datoMes_t mes;

  fecha.dia = 0;
  scanf( "%d", &fecha.dia );
  if (fecha.dia != 0) {
    LeerMes( mes );
    if (mes.esNumero) {
      fecha.mes = mes.dato.mesNumero;
    } else {
      fecha.mes = NumeroDelMes( mes.dato.mesLetra );
    }
    scanf( "%d", &fecha.anno );
    if (fecha.anno < 100) {
      fecha.anno = 2000 + fecha.anno;
    }
  }
}

/*=========================================
  Comparar dos fechas. Devuelve cierto si
  la primera es posterior a la segunda
  =========================================*/
bool EsPosterior( const fecha_t f1, const fecha_t f2 ) {
  if (f1.anno != f2.anno) {
    return (f1.anno > f2.anno);
  } else if (f1.mes != f2.mes) {
    return (f1.mes > f2.mes);
  } else {
    return (f1.dia > f2.dia);
```

```
  }
}

/*===================================================
  Insertar una fecha, en orden, en la lista de fechas
  ===================================================*/
void InsertarFecha( const fecha_t f ) {
  int izquierda, derecha, centro;

  /*-- Poner la fecha como centinela, al final --*/
  lista[numFechas] = f;
  numFechas++;

  /*-- Determinar la posición que le corresponde --*/
  izquierda = 0;
  derecha = numFechas-1;
  while (izquierda < derecha) {
    centro = (izquierda+derecha)/2;
    if (EsPosterior( f, lista[centro] )) {
      izquierda = centro+1;
    } else {
      derecha = centro;
    }
  }

  /*-- Hacer sitio para la nueva fecha --*/
  for (int pos = numFechas-1; pos >= izquierda; pos--) {
    lista[pos+1] = lista[pos];
  }

  /*-- Colocar la fecha en su sitio --*/
  lista[izquierda] = f;
}

/*=================
  Programa principal
  =================*/
int main() {
  fecha_t fecha;     /* fecha leída */
  bool seguir = true;

  printf( "Fechas leídas:\n" );
  numFechas = 0;
  LeerFecha( fecha );
  seguir = (fecha.dia != 0);
  while (seguir && numFechas < maxFechas) {
```

```
    EscribirFecha( fecha );
    if (EsCorrecta(fecha)) {
        InsertarFecha( fecha );
    } else {
      printf( " ** incorrecta **" );
    }
    printf( "\n" );
    LeerFecha( fecha );
    seguir = (fecha.dia != 0);
  }

  printf( "\nFechas en orden:\n" );
  for (int k=0; k<numFechas; k++) {
    EscribirFecha( lista[k] );
    printf( "\n" );
  }
}
```

### 12.7.3 Ejemplo: Recortar una imagen

Este programa es un ejemplo de utilización de una *matriz orlada*. El procedimiento **Recortar** ya fue explicado en el apartado dedicado a las matrices orladas. El procedimiento **Imprimir** es simplemente un recorrido de la matriz para imprimir la imagen contenida. El procedimiento **LeerImagen** lee una imagen de los datos de entrada, leyendo un carácter por cada punto.

El listado completo del programa es el siguiente:

```
/************************************************
* Programa: Recorte
*
* Descripcion:
*   Este programa recorta una imagen digitalizada,
*   es decir, delimita el contorno
************************************************/
#include <stdio.h>

const int Ancho  = 40;  /* anchura de la imagen */
const int Alto   = 20;  /* altura de la imagen */
const int Borde  = -1;  /* marca de borde de la imagen ' ' */
const int Blanco = 0;   /* nivel bajo de gris = blanco '.' */
const int Negro  = 5;   /* nivel alto de gris = negro  '#' */

typedef int Imagen_t[Alto+2][Ancho+2];  /* matriz orlada */
```

```
/** Leer la imagen */
void LeerImagen( Imagen_t imagen ) {
  char c;

  /*-- 1º Paso: Inicializar toda la imagen a "Borde" --*/
  for (int i=0; i<Alto+2; i++) {
    for (int j=0; j<Ancho+2; j++) {
      imagen[i][j] = Borde;
    }
  }

  /*-- 2º Paso: Leer los datos, punto a punto --*/
  for (int i=1; i<=Alto; i++) {
    for (int j=1; j<=Ancho; j++) {
      scanf( " %c", &c );
      imagen[i][j] = int(c) - int('0');
    }
  }
}

/** Contrastar la imagen */
void Contrastar( Imagen_t imagen, int nivel ) {
  for (int i=1; i<=Alto; i++) {
    for (int j=1; j<=Ancho; j++) {
      if (imagen[i][j] <= nivel) {
        imagen[i][j] = Blanco;
      } else {
        imagen[i][j] = Negro;
      }
    }
  }
}

/** Recortar la imagen */
void Recortar( Imagen_t imagen ) {
  bool seguir;

  do { /* fuerza bruta, marcar borde hasta que no cambie más */
    seguir = false;
    for (int i=1; i<=Alto; i++) {
      for (int j=1; j<=Ancho; j++) {
        if (imagen[i][j] == Blanco && (
              imagen[i-1][j] == Borde ||
              imagen[i][j-1] == Borde ||
              imagen[i][j+1] == Borde ||
```

```
              imagen[i+1][j] == Borde )
            ) {
          imagen[i][j] = Borde;
          seguir = true;
        }
      }
    }
  } while (seguir);
}

/** Imprimir la imagen */
void Imprimir( const Imagen_t imagen ) {
  const char Punto[9] = {' ', '.', ':', '+', 'x', '*', '#'};

  for (int i=1; i<=Alto; i++) {
    for (int j=1; j<=Ancho; j++) {
      printf( "%c", Punto[imagen[i][j]+1] );
    }
    printf( "\n" );
  }
}

/** Programa principal */
int main() {
  Imagen_t imagen;

  /*-- Leer la imagen inicial --*/
  LeerImagen( imagen );
  printf( "Imagen inicial:\n" );
  Imprimir( imagen );

  /*-- Reducir la imagen a blanco y negro --*/
  Contrastar( imagen, 3 );
  printf( "\nImagen contrastada:\n" );
  Imprimir( imagen );

  /*-- Recortar la imagen, marcando el borde externo --*/
  Recortar( imagen );
  printf( "\nImagen recortada:\n" );
  Imprimir( imagen );
}
```

A continuación se presenta un ejemplo de la ejecución de este programa. Los datos de entrada contienen un carácter por cada punto de la imagen, y simulan una rejilla. Una de las esquinas del borde derecho está rota. Los datos son:

```
0123454321012345432101234543210123454321
1234545432123454543212345454321234545432
2345434543234543454323454345432345434543
3454323454345432345434543234543454323454
4543212345454321234545432123454543212344
5432101234543210123454321012345432101233
4543212345454321234545432123454543212344
3454323454345432345434543234543454323454
2345434543234543454323454345432345434543
1234545432123454543212345454321234545432
0123454321012345432101234543210123454321
1234545432123454543212345454321234545432
2345434543234543454323454345432345434543
3454323454345432345434543234543454323454
4543212345454321234545432123454543212345
5432101234543210123454321012345432101234
4543212345454321234545432123454543212345
3454323454345432345434543234543454323454
2345434543234543454323454345432345434543
1234545432123454543212345454321234545432
```

El resultado de la ejecución del programa es el siguiente:

```
Imagen inicial:
.:+x*#*x+:.:+x*#*x+:.:+x*#*x+:.:+x*#*x+:
:+x*#*#*x+:+x*#*#*x+:+x*#*#*x+:+x*#*#*x+
+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x
x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*
*#*x+:+x*#*#*x+:+x*#*#*x+:+x*#*#*x+:+x**
#*x+:.:+x*#*x+:.:+x*#*x+:.:+x*#*x+:.:+xx
*#*x+:+x*#*#*x+:+x*#*#*x+:+x*#*#*x+:+x**
x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*
+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x
:+x*#*#*x+:+x*#*#*x+:+x*#*#*x+:+x*#*#*x+
.:+x*#*x+:.:+x*#*x+:.:+x*#*x+:.:+x*#*x+:
:+x*#*#*x+:+x*#*#*x+:+x*#*#*x+:+x*#*#*x+
+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x
x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*
*#*x+:+x*#*#*x+:+x*#*#*x+:+x*#*#*x+:+x*#
#*x+:.:+x*#*x+:.:+x*#*x+:.:+x*#*x+:.:+x*
*#*x+:+x*#*#*x+:+x*#*#*x+:+x*#*#*x+:+x*#
x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*
+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x+x*#*x*#*x
:+x*#*#*x+:+x*#*#*x+:+x*#*#*x+:+x*#*#*x+
```

```
Imagen contrastada:
....###.......###.......###.......###...
...#####.....#####.....#####.....#####..
..###.###...###.###...###.###...###.###.
.###...###.###...###.###...###.###...###
###.....#####.....#####.....#####.....##
##.......###.......###.......###........
###.....#####.....#####.....#####.....##
.###...###.###...###.###...###.###...###
..###.###...###.###...###.###...###.###.
...#####.....#####.....#####.....#####..
....###.......###.......###.......###...
...#####.....#####.....#####.....#####..
..###.###...###.###...###.###...###.###.
.###...###.###...###.###...###.###...###
###.....#####.....#####.....#####.....##
##.......###.......###.......###.......#
###.....#####.....#####.....#####.....##
.###...###.###...###.###...###.###...###
..###.###...###.###...###.###...###.###.
...#####.....#####.....#####.....#####..

Imagen recortada:
    ###       ###       ###       ###
   #####     #####     #####     #####
  ###.###   ###.###   ###.###   ### ###
 ###...### ###...### ###...### ###   ###
###.....#####.....#####.....#####     ##
##.......###.......###.......###
###.....#####.....#####.....#####     ##
 ###...###.###...###.###...###.###   ###
  ###.###...###.###...###.###...### ###
   #####.....#####.....#####.....#####
    ###.......###.......###.......###
   #####.....#####.....#####.....#####
  ###.###...###.###...###.###...###.###
 ###...###.###...###.###...###.###...###
###.....#####.....#####.....#####.....##
##.......###.......###.......###.......#
###.....#####.....#####.....#####.....##
 ###...### ###...### ###...### ###...###
  ###.###   ###.###   ###.###   ###.###
   #####     #####     #####     #####
```

Como puede verse, la marca de borde exterior se ha ido propagando a todos los puntos vecinos que estaban en blanco, hasta rellenar toda la zona exterior de la rejilla, incluyendo la malla abierta.

# Tema 13

# Punteros y variables dinámicas

En este tema se introducen estructuras de datos potencialmente ilimitadas. Se justifica su interés y se describe en particular la estructura secuencia.

## 13.1 Estructuras de datos no acotadas

En los temas anteriores se han descrito las estructuras de datos que pueden definirse en **C±**. Todas ellas tienen una característica en común: la capacidad total (número de elementos componentes) se determina explícitamente al definirlas. Una estructura podrá usarse de manera que el número de componentes que contengan información significativa sea variable, pero nunca mayor que el tamaño total de la estructura.

Por ejemplo, en el tema anterior se ha presentado un programa para escribir en orden una serie de fechas leídas como datos. El número de fechas leídas cambia de una ejecución del programa a otra, pero la capacidad total del programa viene limitada por el tamaño de la estructura definida para almacenar la lista de fechas.

La fijación del tamaño máximo representa una solución de compromiso entre la capacidad del programa y su eficiencia. Si el tamaño es relativamente pequeño el programa tendrá una capacidad de tratamiento limitada. Si el tamaño se fija a un valor muy grande, el programa será poco eficiente en el uso de la memoria, pues necesitará espacio para toda la estructura de datos, aunque sólo se aproveche una pequeña parte.

Tras el análisis anterior debe resultar evidente que sería útil disponer de estructuras de datos que no tuvieran un tamaño fijado de antemano, sino que

pudieran ir creciendo o reduciendo su tamaño en función de los datos particulares que se estén manejando en cada ejecución del programa. Estas estructuras de datos se denominan, en general, *estructuras dinámicas*, y poseen la cualidad de que su tamaño es potencialmente ilimitado, aunque, naturalmente, no podrá exceder la capacidad física del computador que ejecute el programa.

## 13.2 La estructura secuencia

La estructura *secuencia* puede definirse como un esquema de datos del tipo iterativo, pero con un número variable de componentes. La estructura secuencia resulta parecida a una formación con número variable de elementos.

En realidad existen diferentes esquemas secuenciales de datos. Aun teniendo en común que el número de elementos pueda variar, hay varias formas posibles de plantear las operaciones sobre secuencias. Para describir las distintas alternativas distinguiremos entre operaciones de construcción y de acceso. Con las primeras podremos añadir o eliminar componentes de la secuencia. Con las segundas podremos obtener o modificar el valor de las componentes que existen en un momento dado.

Las operaciones de construcción pueden incluir:

- Añadir o retirar componentes al principio de la secuencia.
- Añadir o retirar componentes al final de la secuencia.
- Añadir o retirar componentes en posiciones intermedias de la secuencia.

Las operaciones de acceso pueden ser:

*Acceso secuencial*: Las componentes deben tratarse una por una, en el orden en que aparecen en la secuencia.

*Acceso directo*: Se puede acceder a cualquier componente directamente indicando su posición, como en una formación o vector.

En este tema se presentan varias estructuras de datos utilizables en **C±** y que responden al esquema secuencia, con distintas posibilidades, según los casos.

En muchos casos, y en particular cuando el acceso es secuencial, el tratamiento de una secuencia se realiza empleando un *cursor*. El cursor es una variable que señala a un elemento de la secuencia. El acceso, inserción o eliminación de componentes de la secuencia se hace actuando sobre el elemento señalado por el cursor. Dicho elemento lo representaremos simbólicamente como *cursor*$^{\uparrow}$, empleando la flecha ($^{\uparrow}$) como símbolo gráfico para designar el elemento de información señalado por otro. Para actuar sobre el cursor se suelen plantear las siguientes operaciones:

- *Iniciar*: Pone el cursor señalando al primer elemento.
- *Avanzar*: El cursor pasa a señalar al siguiente elemento.
- *Fin*: Es una función que indica si el cursor ha llegado al final de la secuencia.

El empleo de un cursor se ilustra en la figura 13.1.

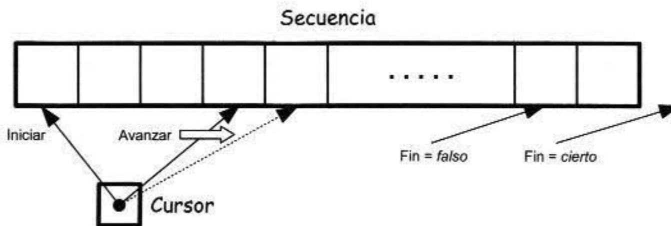



Figura 13.1 Manejo de una secuencia mediante cursor.

Por el momento, ilustraremos el concepto de secuencia describiendo de forma abstracta un programa para leer números enteros y escribirlos en orden, usando una secuencia ilimitada para almacenar los números leídos. Supondremos que el final de los datos se indica mediante un valor cero. Los primeros pasos de refinamiento del programa serían:

*Leer números e imprimirlos en orden* ⟶
    *Leer los números y almacenarlos en orden*
    *Imprimir los números almacenados*

*Leer los números y almacenarlos en orden* ⟶

```
Iniciar la secuencia, vacía
Leer un número
while (no es cero) {
  Almacenar el número, en orden
  Leer un número
}
```

*Almacenar el número, en orden* ⟶
    *Buscar la posición que le corresponde*
    *Insertarlo en su posición*

*Buscar la posición que le corresponde* ⟶

```
Iniciar cursor
while (no Fin && número > cursor↑) {
  Avanzar cursor
}
```

*Imprimir los números almacenados* ⟶

```
Iniciar cursor
while (no Fin) {
   Imprimir cursor↑
   Avanzar cursor
}
```

En estos últimos refinamientos se ha usado la técnica del cursor para localizar la posición en la secuencia en que deberá insertarse el nuevo número, y para imprimir la secuencia de números. Obsérvese que se han empleado las operaciones básicas indicadas anteriormente: *Iniciar* el cursor al comienzo, *Avanzar* el cursor, y detectar si se llega al *Fin* de la secuencia. También se ha usado *cursor*↑ para hacer referencia a la componente de la secuencia accesible en cada momento.

En este ejemplo se supone, además, que es posible insertar una nueva componente en medio de la secuencia. El desarrollo completo del programa en **C±** se describe más adelante.

## 13.3 Variables dinámicas

Una manera de realizar estructuras de datos ilimitadas en **C±** es mediante el empleo de *variables dinámicas*. Una variable dinámica no se declara como tal, sino que se crea en el momento necesario, y se destruye cuando ya no se necesita. Las variables dinámicas no tienen nombre, sino que se designan mediante otras variables llamadas punteros o referencias.

### 13.3.1 Punteros

En **C±** los *punteros* o *referencias* son variables simples cuyo contenido es precisamente una referencia a otra variable. El valor de un puntero no es representable como número o texto. En su lugar usaremos una representación gráfica en la que utilizaremos una flecha para enlazar una variable de tipo puntero con la variable a la que hace referencia, tal como se indica en la figura 13.2.

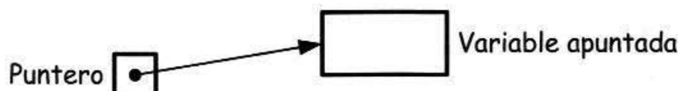



Figura 13.2 Puntero y su variable apuntada.

Los punteros de **C±** están tipados, al igual que los demás valores manejados en el lenguaje. El tipo de un puntero especifica en realidad el tipo de variable a la que puede apuntar. La declaración es:

```
typedef Tipo-de-variable* Tipo-puntero;
```

Una vez declarado el tipo, se pueden declarar variables puntero de dicho tipo. Una variable puntero se puede usar para designar la variable apuntada mediante la notación:

```
*puntero
```

Por ejemplo:

```
typedef int* Tp_Entero;
Tp_Entero pe;

*pe = 33;
printf( "%d", *pe );
```

Estas sentencias asignan el valor 33 a la variable dinámica señalada por el puntero `pe`, y luego la imprimen. Para que estas sentencias funcionen correctamente es necesario que exista realmente la variable apuntada. Si el puntero no señala realmente a una variable dinámica, el resultado de usar **puntero* será imprevisible.

Para poder detectar si un puntero señala realmente o no a otra variable, existe en **C±** el valor especial `NULL` (que no es una palabra clave, sino que está definido en la librería estándar **`stdlib.h`** y también en otras librerías). Este valor es compatible con cualquier tipo de puntero, e indica que el puntero no señala a ninguna parte. Por lo tanto debería ser asignado a cualquier puntero que sepamos que no señala a ninguna variable. Normalmente se usará para inicializar las variables de tipo puntero al comienzo del programa. La inicialización no es automática, sino que debe ser realizada expresamente por el programador. Por ejemplo:

```
if (pe != NULL) {
  *pe = 33;
  printf( "%d", *pe );
}
```

En principio esta sentencia garantizaría que sólo se usa la variable apuntada cuando realmente existe. En realidad eso no es del todo cierto, ya que sólo una correcta disciplina en el uso de punteros permite asumir que sólo tienen valor no nulo los punteros que realmente señalan a variables que existen. El lenguaje **C±** en sí mismo no puede garantizarlo (es una limitación inherente a C

y C++). Por ejemplo, se puede destruir una variable dinámica pero conservar punteros que la referenciaban, y que ahora señalan a algo inexistente.

## 13.3.2 Uso de variables dinámicas

Una variable de un programa se corresponde, en general, con una zona concreta de la memoria que el compilador reserva para almacenar en ella el valor de la variable. Las variables normales de un programa tienen esa zona de memoria reservada de antemano al empezar a ejecutarse el programa o subprograma en que se declaran, y por tanto pueden ser usadas en cualquier momento. Conviene recordar que las variables declaradas en un subprograma sólo existen mientras se ejecutan las sentencias de ese subprograma.

Las variables dinámicas no tienen ese espacio de memoria reservado de antemano, sino que se crean a partir de punteros en el momento en que se indique. Además, una vez creadas siguen existiendo incluso después de que termine la ejecución del subprograma donde se crean. La forma más sencilla de crear una variable dinámica es mediante el operador **new**:

```
typedef Tipo-de-variable* Tipo-puntero;
Tipo-puntero puntero;

puntero = new Tipo-de-variable;
```

El operador **new** crea una variable dinámica del tipo indicado y devuelve una referencia que puede asignarse a un puntero de tipo compatible. Como en cualquier otra asignación, el valor anterior del puntero se pierde. Tal como se ha dicho antes, la variable dinámica no tiene nombre, y sólo se puede hacer referencia a ella a través del puntero. Podemos representar gráficamente el efecto de esta sentencia según se muestra en la figura 13.3.

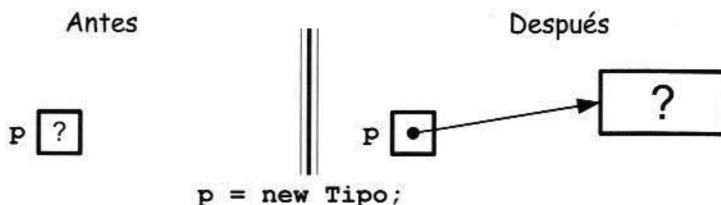



Figura 13.3 Creación de una variable dinámica.

La variable dinámica se crea a base de reservarle el espacio necesario en una zona general de memoria gestionada dinámicamente. En principio no se puede asumir que la variable recién creada tenga un valor concreto, igual que las variables normales que se declaran sin un valor inicial explícito.

Las variables dinámicas, una vez creadas, siguen existiendo hasta que se indique explícitamente que ya no son necesarias, en cuyo caso el espacio que se había reservado para ellas quedará otra vez disponible para crear nuevas variables dinámicas. Para ello existe la sentencia `delete`, que permite destruir la variable dinámica a la que señala un puntero:

```
delete puntero;
```

Esta sentencia destruye la variable apuntada pero no garantiza que el puntero quede con un valor determinado. En particular no garantiza que tome valor NULL.

Una variable dinámica puede estar referenciada por más de un puntero. Esto ocurre cuando se copia un puntero en otro. Por ejemplo:

```
typedef int* Tp_Entero;
Tp_Entero  p1, p2;

p1 = new int;
p2 = p1;
```

Gráficamente en la figura 13.4 se muestra el resultado de copiar un puntero en otro.

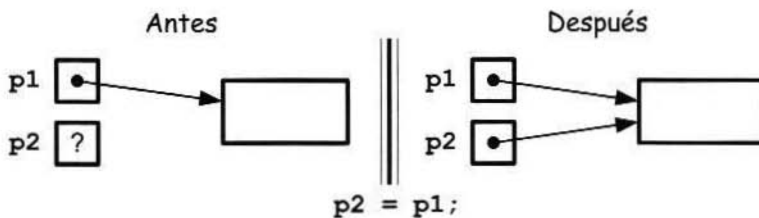



Figura 13.4 Copia de un puntero en otro.

Tanto la variable `p1` como `p2` quedan señalando a la misma variable dinámica.

Un problema delicado al manejar variables dinámicas es que pueden quedar perdidas, sin posibilidad de hacer referencia a ellas. Esto ocurre en el siguiente ejemplo:

```
typedef int* Tp_Entero;
Tp_Entero  p1, p2;

p1 = new int;
p2 = new int;
p2 = p1;
```

En la figura 13.5 se muestra gráficamente el resultado.

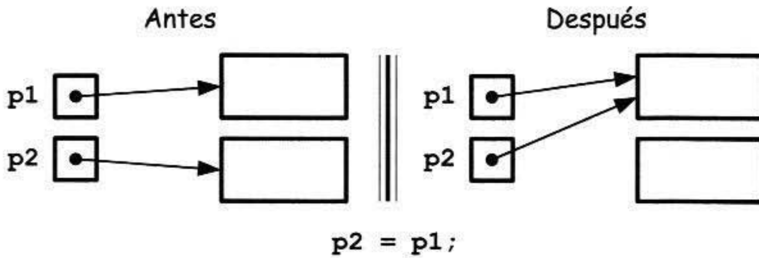



Figura 13.5 Variable dinámica perdida.

En este caso la variable creada mediante `p2 = new int;` queda perdida, sin posibilidad de ser usada, ya que las variables dinámicas no tienen nombre, y el único puntero que la señalaba ha cambiado su valor, perdiendo el anterior. Además el espacio ocupado por la variable dinámica sigue reservado, lo cual es totalmente inútil y representa una pérdida de la capacidad de memoria disponible (en inglés se denomina *memory leak*).

## 13.4 Realización de secuencias mediante punteros

Los punteros son un elemento de programación de muy bajo nivel. Los lenguajes de programación simbólicos deberían evitar su empleo, sustituyéndolo por mecanismos más potentes de declaración de estructuras de datos, que permitiesen definir directamente estructuras dinámicas ilimitadas.

Desgraciadamente, muchos lenguajes están diseñados pensando en que su compilación no sea demasiado complicada. Las estructuras de datos con tamaño variable presentan algunas complicaciones para ser manejadas de manera eficiente, y es frecuente que los lenguajes de programación no incorporen directamente esquemas de datos de tamaño variable. Esta limitación facilita el trabajo de compilación, ya que todas las variables tendrán un tamaño fijo que puede ser calculado por el compilador, y determinar así el espacio de memoria que ha de reservarse para cada una.

**C±** no dispone de esquemas de datos de tamaño variable. Dichos esquemas pueden ser realizados indirectamente por el programador mediante el uso de punteros. Al hacerlo convendrá tener cuidado, y emplearlos de una manera precisa, traduciendo a punteros los mecanismos de definición de alto nivel que deberían estar disponibles.

La definición simbólica de una estructura ilimitada basándose en esquemas con número fijo de elementos será, normalmente, recursiva. Una definición

*recursiva* es aquella en que se hace referencia a sí misma. Sería deseable que una secuencia ilimitada se pudiese definir de manera recursiva, sin necesidad de punteros, de una forma parecida a la siguiente:

```
typedef struct Tipo-secuencia {
  bool vacia;  /* indica si es la secuencia vacía */
  Tipo-componente primero;  /* sólo si no es vacía */
  Tipo-secuencia resto;     /* sólo si no es vacía */  /* ERROR */
};
```

Esta definición nos dice que una secuencia ilimitada de componentes es una de dos cosas posibles: o bien una secuencia vacía, o bien una primera componente seguida de la secuencia formada por el resto de las componentes.

Lamentablemente esta forma de definición recursiva no es admisible en **C±**. Para definir una secuencia ilimitada tendremos que recurrir al empleo de variables dinámicas y punteros. Una manera de hacerlo es usar punteros para enlazar cada elemento de la secuencia con el siguiente tal y como se muestra en la figura 13.6.

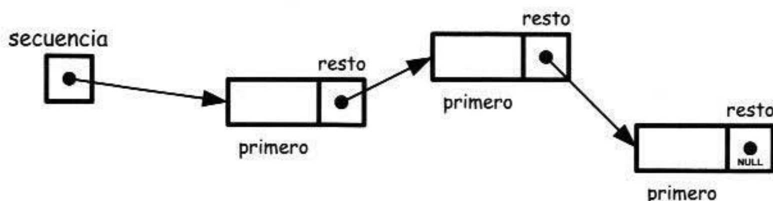



Figura 13.6 Secuencia enlazada mediante punteros.

Cada elemento de la secuencia se materializa como un registro con dos campos: el primero contiene el valor de una componente, y el segundo es un puntero que señala al siguiente. El último elemento tendrá el puntero al siguiente con valor `NULL`. La secuencia completa es accesible a través de un puntero que señala al comienzo de la misma.

Aplicando este esquema con punteros sí es posible definir una secuencia ilimitada en **C±**. La siguiente definición trata de ser lo más parecida posible a la definición recursiva propuesta antes:

```
typedef struct Tipo-nodo {
  Tipo-componente primero;
  Tipo-nodo * resto;
};

typedef Tipo-nodo * Tipo-secuencia;
```

Esta pareja de definiciones es válida en **C±**, aunque tiene una característica excepcional: se usa el identificador *Tipo_nodo* antes de haber sido definido completamente. Esto sólo es posible hacerlo en declaraciones de punteros como la anterior. Gracias a esa posibilidad se puede realizar mediante punteros algo equivalente a una definición recursiva de un esquema de datos.

Una vez definidos los tipos de la secuencia y sus componentes se podrán declarar variables de dichos tipos y operar con ellos:

```
Tipo-componente valor;
Tipo-secuencia secuencia, siguiente;

if (secuencia != NULL) {
  (*secuencia).primero = valor;
  siguiente = (*secuencia).resto;
}
```

La combinación del operador de desreferenciación de puntero (`*`) y la selección de campo de registro (`.`) es incómoda de escribir, porque requiere paréntesis, y difícil de leer. Por esta razón **C±** permite combinar ambos en un operador único con una grafía más amigable (`->`). Las sentencias anteriores se pueden reescribir de la forma siguiente, mucho más fácil de leer:

```
if (secuencia != NULL) {
  secuencia->primero = valor;
  siguiente = secuencia->resto;
}
```

### 13.4.1 Operaciones con secuencias enlazadas

Describiremos la manera de realizar algunas operaciones típicas sobre secuencias enlazadas con punteros. En ellas supondremos la existencia de un cursor que va señalando a las componentes una tras otra. El cursor será simplemente un puntero. Como ejemplos de operaciones desarrollaremos algunos fragmentos de un programa para leer números enteros e imprimirlos en orden, así como para quitar luego de la lista ordenada los números que se indiquen.

**DEFINICIÓN** - La definición de la secuencia será:

```
typedef struct TipoNodo {
  int valor;
  TipoNodo * siguiente;
};
typedef TipoNodo * TipoSecuencia;
TipoSecuencia secuencia;
```

**RECORRIDO** - El recorrido de toda la secuencia se consigue mediante un bucle de acceso a elementos y avance del cursor. Puesto que la secuencia tiene un número indefinido de elementos, no se usará un bucle con contador. En este caso usaremos un esquema `while` . Como ejemplo describimos la escritura de los valores de la secuencia.

```
typedef TipoNodo * TipoPuntNodo;
TipoPuntNodo cursor;

cursor = secuencia;
while (cursor != NULL) {
  printf( "%5d", cursor->valor );
  cursor = cursor->siguiente;
}
```

**BÚSQUEDA** - La búsqueda en una secuencia enlazada ha de hacerse de forma secuencial. La búsqueda es parecida al recorrido, pero la condición de terminación cambiará. De hecho habrá una doble condición de terminación: que se localice el elemento buscado, y/o que se agote la secuencia. A continuación se presenta la búsqueda de la posición en que ha de insertarse un nuevo número en la secuencia ordenada. La posición será la que ocupe el primer valor igual o mayor que el que se quiere insertar.

```
int numero;   /* valor a buscar */
TipoPuntNodo cursor, anterior;

cursor = secuencia;
anterior = NULL;
while (cursor != NULL && cursor->valor < numero) {
  anterior = cursor;
  cursor = cursor->siguiente;
}
```

Al salir del bucle `cursor` queda señalando al punto en que deberá insertarse el nuevo elemento, y `anterior` señala al elemento que lo precede. Esto resulta útil para realizar luego operaciones de inserción o borrado, como se verá a continuación.

**INSERCIÓN** - La inserción de un nuevo elemento se consigue creando una variable dinámica para contenerlo, y modificando los punteros para enlazar dicha variable dentro de la secuencia. El caso más sencillo es el de insertar un nuevo elemento detrás de uno dado. La representación gráfica se muestra en la figura 13.7

En la figura, el nuevo elemento creado tiene un fondo diferente, además se han marcado con línea discontinua los enlaces creados o modificados por estas

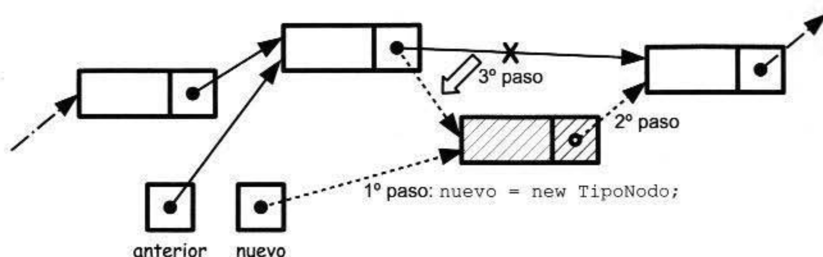



Figura 13.7 Inserción en una secuencia de punteros.

operaciones. El orden de las operaciones a realizar resulta esencial para que no se produzca la pérdida de ninguna variable dinámica y por ello, en la figura también se ha detallado el orden de los pasos a realizar. El código en **C±** será:

```
int numero;   /* valor a insertar */
TipoPuntNodo cursor, anterior, nuevo;

nuevo = new TipoNodo;                      /* 1º paso */
nuevo->valor = numero;
nuevo->siguiente = anterior->siguiente; /* 2º paso */
anterior->siguiente = nuevo;            /* 3º paso */
```

**BORRADO** - Para borrar un elemento hay que quitar el nodo que lo contiene, enlazando directamente el anterior con el siguiente tal como se indica en la figura 13.8. Es la operación inversa de la inserción. Si el nodo que contenía el elemento ya no es necesario hay que destruirlo explícitamente. También en este caso es importante seguir el orden que se detalla en la figura 13.8.

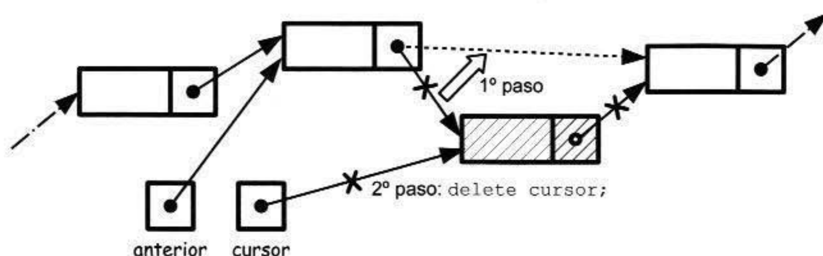



Figura 13.8 Borrado en una secuencia de punteros.

Igual que antes, en la figura 13.8 se han marcado con línea discontinua los enlaces modificados por las operaciones de borrado. Además, el elemento borrado aparece con un fondo diferente. Para hacer el código más robusto se ha forzado el **cursor** a valor nulo, ya que de no hacerlo así quedaría apuntando

a un lugar que ya no existe (marcado con una X en la figura). El código en **C±** será:

```
TipoPuntNodo cursor, anterior;

anterior->siguiente = cursor->siguiente;
delete cursor;
cursor = NULL;
```

Estos ejemplos de inserciones y borrados corresponden al caso general y operan con elementos en medio de la secuencia. Si la inserción o borrado debe hacerse al principio de la secuencia el código es algo diferente, ya que no hay elemento anterior, pero es igualmente sencillo.

### 13.4.2 Ejemplo: Leer números y escribirlos en orden

Reuniendo los fragmentos de código anteriores se puede escribir ya este ejemplo. Aquí se ha tenido en cuenta el caso especial de insertar o borrar al principio de la secuencia. Esta situación se detecta porque no existe elemento anterior, y el puntero al elemento `anterior` queda con valor `NULL` después de la búsqueda.

El resto del código del programa es esencialmente idéntico a los fragmentos desarrollados en los ejemplos anteriores. La mayor parte del código añadido corresponde a la lectura de los datos y la escritura de los resultados. El listado del programa completo es el siguiente:

```
/****************************************************
* Programa: Secuencia
*
* Descripción:
*   Programa que lee una serie de números enteros,
*   los almacena en una secuencia enlazada, en orden,
*   y los imprime. Después se pueden borrar de
*   manera selectiva.
****************************************************/
#include <stdio.h>

typedef struct TipoNodo {
  int valor;
  TipoNodo * siguiente;
};
typedef TipoNodo * TipoSecuencia;
typedef TipoNodo * TipoPuntNodo;
```

```
int main() {
  int numero;
  TipoSecuencia secuencia;
  TipoPuntNodo cursor, anterior, nuevo;

  /*-- Leer los datos y almacenarlos --*/
  printf( "Datos leídos:\n" );
  secuencia = NULL;
  scanf( "%d", &numero );   /* primer dato */
  while (numero != 0) {
    printf( " %d", numero );

    /*-- Buscar posición para el nuevo número --*/
    /* 'anterior' señalará al nodo detrás del
       cual hay que insertar el nuevo valor */
    cursor = secuencia;
    anterior = NULL;
    while (cursor != NULL && cursor->valor < numero) {
      anterior = cursor;
      cursor = cursor->siguiente;
    }

    /*-- Crear nodo con el nuevo número --*/
    nuevo = new TipoNodo;
    nuevo->valor = numero;

    /*-- Insertar el nodo en la secuencia --*/
    if (anterior == NULL) {
      /*-- Insertar al comienzo de la secuencia --*/
      nuevo->siguiente = secuencia;
      secuencia = nuevo;
    } else {
      /*-- Insertar detrás del anterior --*/
      nuevo->siguiente = anterior->siguiente;
      anterior->siguiente = nuevo;
    }
    scanf( "%d", &numero );   /* siguiente dato */
  }
  printf( "\n" );

  /*-- Mostrar la lista ya ordenada --*/
  printf( "\nDatos ordenados\n" );
  cursor = secuencia;
  while (cursor != NULL) {
    printf( "%5d", cursor->valor );
```

```
    cursor = cursor->siguiente;
  }
  printf( "\n" );

  /*-- Bucle de búsqueda y borrado de números --*/
  printf( "\nDatos a borrar:\n" );
  scanf( "%d", &numero );     /* primer dato */
  while (numero != 0) {

    /*-- Buscar posición del número a borrar --*/
    cursor = secuencia;
    anterior = NULL;
    while (cursor != NULL && cursor->valor != numero) {
      anterior = cursor;
      cursor = cursor->siguiente;
    }

    /*-- Borrar el número encontrado --*/
    if (cursor != NULL) {  /* el número está en la lista */
      if (anterior != NULL) {
        anterior->siguiente = cursor->siguiente;
      } else {
        secuencia = cursor->siguiente;
      };
      delete cursor;
      printf( " %d borrado\n", numero );
      cursor = secuencia;
      while (cursor != NULL) {
        printf( "%5d", cursor->valor );
        cursor = cursor->siguiente;
      }
      printf( "\n" );
    } else {             /* el número no está en la lista */
      printf( " %d no encontrado\n", numero );
    }

    scanf( "%d", &numero );   /* siguiente dato */
  }
}
```

La ejecución del programa utilizando los siguientes datos de ejemplo:

```
12 34 2 56 4 7 1 33 0
4 11 1 56 7 0
```

proporciona los siguientes resultados:

```
Datos leídos:
 12 34 2 56 4 7 1 33

Datos ordenados
    1    2    4    7   12   33   34   56

Datos a borrar:
 4 borrado
    1    2    7   12   33   34   56
 11 no encontrado
 1 borrado
    2    7   12   33   34   56
 56 borrado
    2    7   12   33   34
 7 borrado
    2   12   33   34
```

## 13.5 Punteros y paso de argumentos

El manejo de punteros cuando se utilizan como argumentos de un subprograma tiene ciertas peculiaridades que requieren un estudio más detallado.

### 13.5.1 Paso de punteros como argumentos

Como cualquier otro dato, un puntero puede pasarse como argumento a un subprograma. Así, la operación de imprimir la lista de números enteros del ejemplo anterior podría redactarse como procedimiento que reciba como argumento la secuencia enlazada. Tal como se veía, una secuencia enlazada se maneja a partir del puntero al primer elemento. El código del procedimiento y un ejemplo de cómo invocarlo sería:

```
void ImprimirLista( TipoSecuencia lista ) {
  TipoPuntNodo cursor = lista;

  while (cursor != NULL) {
    printf( "%5d", cursor->valor );
    cursor = cursor->siguiente;
  }
  printf( "\n" );
}
```

```
TipoSecuencia secuencia;

ImprimirLista( secuencia );
```

Por defecto, los datos de tipo puntero se pasan como argumentos por valor. Es lo que ocurre en el ejemplo anterior. Si se desea usar un subprograma para modificar datos de tipo puntero, entonces habrá que pasar el puntero por referencia. Por ejemplo, si planteamos como subprograma la operación de búsqueda en una secuencia enlazada podríamos escribir:

```
void Buscar( TipoSecuencia lista, int numero,
             TipoPuntNodo & cursor, TipoPuntNodo & anterior ) {

  cursor = lista;
  anterior = NULL;
  while (cursor != NULL && cursor->valor != numero) {
    anterior = cursor;
    cursor = cursor->siguiente;
  }
}

TipoSecuencia secuencia;
TipoPuntNodo encontrado, previo;
int dato;

Buscar( secuencia, dato, encontrado, previo );
```

En la llamada a `Buscar` la variable `secuencia` no podrá ser modificada, ya que el argumento `lista` se pasa por valor. Por el contrario, las variables de tipo puntero `encontrado` y `previo` serán modificadas por el subprograma para reflejar el resultado de la búsqueda.

### 13.5.2 Paso de argumentos mediante punteros

En general el valor de un puntero en sí mismo no es significativo, sino que el puntero es sólo un medio para designar la variable apuntada. Desde un punto de vista conceptual el paso de un puntero como argumento puede ser considerado equivalente a pasar como argumento la variable apuntada.

Por ejemplo, si queremos pasar como argumento una variable dinámica podemos recurrir a un puntero como elemento intermedio para designarla. Esto representa una dificultad añadida para entender cómo funciona un determinado fragmento de código, y de hecho representa un peligro potencial de cometer determinados errores de codificación.

La dificultad reside en el hecho de que pasar un puntero por valor no evita que el subprograma pueda modificar la variable apuntada. Por ejemplo:

```
typedef int* Tp_Entero;

void Imprimir( Tp_Entero val ) {
  printf( "%d", *val );
}

void Incrementar( Tp_Entero val ) {
  *val = *val + 1;
}

Tp_Entero  p1;

p1 = new int;
Imprimir( p1 );
Incrementar( p1 );
```

Tanto el procedimiento de `Imprimir` como el de `Incrementar` reciben un puntero pasado por valor. El primero no modifica la variable apuntada pero el segundo sí. Al establecer la analogía entre el paso como argumento del puntero y el paso como argumento de la variable apuntada, la distinción entre paso por valor y por referencia se pierde. El paso por valor de un puntero equivale al paso por referencia de la variable apuntada.

En realidad los punteros se usan implícitamente para pasar argumentos por referencia. Cuando se declara un argumento pasado por referencia, lo que hace realmente el compilador es pasar un puntero a la variable externa usada como argumento actual en la llamada. Dado el subprograma:

```
void Duplicar( int & valor ) {
  valor = 2 * valor;
}
```

El código interno generado por el compilador es equivalente a:

```
typedef int * P_int;

void Duplicar( P_int p_valor ) {
  *p_valor = 2 * (*p_valor);
}
```

De hecho la notación & para indicar el paso de argumento por referencia es una mejora de C++ respecto a C. En lenguaje C hay que usar siempre un puntero explícito para pasar argumentos por referencia. Por supuesto, hace

falta entonces un operador especial para obtener el valor de un puntero a una variable (estática) y usarlo en la llamada al subprograma. En lenguaje C ese operador corresponde también al símbolo &, como se muestra a continuación:

```
int numero;

Duplicar( &numero );
```

Ahora debe quedar clara la manera de invocar ciertas funciones estándar de C, tal como `scanf()`. A lo largo de todo este libro han ido apareciendo sentencias de lectura como la siguiente:

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

Lo que se está haciendo en esta llamada es pasar como argumento un puntero que señala a la variable **numero**, a través del cual se puede modificar su valor. Hay que hacer notar que el puntero en sí se está pasando por valor, y eso es equivalente a pasar el dato apuntado por referencia.

En bastantes casos cuando se trabaja con variables dinámicas, que sólo son accesibles a través de punteros, resulta natural usar un puntero explícito para pasar como argumento la variable dinámica apuntada. Insistiremos en que el paso del puntero equivale a pasar la variable apuntada siempre por referencia. En **C±** para pasar la variable apuntada por valor hay que hacerlo de la manera convencional, como en el siguiente ejemplo de código:

```
typedef int* Tp_Entero;

void Imprimir( int val ) {              /* paso por valor */
  printf( "%d", val );
}

void Incrementar( Tp_Entero val ) {  /* paso por referencia */
  *val = *val + 1;
}

Tp_Entero  p1;

p1 = new int;
Imprimir( *p1 );
Incrementar( p1 );
```

NOTA: En lenguaje C++ existe la posibilidad de programar algo equivalente al paso por valor, pero usando un puntero explícito como intermediario. Esto se consigue usando adecuadamente el cualificador `const` en puntos determinados de la declaración del tipo puntero y/o del argumento. Lamentablemente la semántica de estas construcciones es extraordinariamente compleja y confusa, por lo que se ha optado por no incluirlas en el subconjunto **C±**.

### 13.5.3 Ejemplo: Leer números y escribirlos en orden

Como ejemplo del uso de punteros como argumentos se reescribe a continuación el mismo ejemplo de manejo de una secuencia de números, pero definiendo ahora como subprogramas algunas de las operaciones sobre la secuencia. El listado del nuevo programa es el siguiente:

```
/**************************************************
* Programa: Secuencia2
*
* Descripción:
*   Programa que lee una serie de números enteros,
*   los almacena en una secuencia enlazada, en orden,
*   y los imprime. Después se pueden borrar de
*   manera selectiva.
**************************************************/
#include <stdio.h>

typedef struct TipoNodo {
  int valor;
  TipoNodo * siguiente;
};
typedef TipoNodo * TipoSecuencia;
typedef TipoNodo * TipoPuntNodo;

/** Insertar un nuevo número en una secuencia ordenada */
void InsertarEnOrden( TipoSecuencia & secuencia, int numero ) {
  TipoPuntNodo cursor, anterior, nuevo;

  /*-- Buscar posición para el nuevo número --*/
  /* 'anterior' señalará al nodo detrás del
     cual hay que insertar el nuevo valor */
  cursor = secuencia;
  anterior = NULL;
  while (cursor != NULL && cursor->valor < numero) {
    anterior = cursor;
    cursor = cursor->siguiente;
  }

  /*-- Crear nodo con el nuevo número --*/
  nuevo = new TipoNodo;
  nuevo->valor = numero;

  /*-- Insertar el nodo en la secuencia --*/
  if (anterior == NULL) {
```

```
    /*-- Insertar al comienzo de la secuencia --*/
    nuevo->siguiente = secuencia;
    secuencia = nuevo;
  } else {
    /*-- Insertar detrás del anterior --*/
    nuevo->siguiente = anterior->siguiente;
    anterior->siguiente = nuevo;
  }
}

/** Quitar todas la apariciones de un número de una secuencia de números */
void Eliminar( TipoSecuencia & secuencia, int numero, bool & encontrado ) {
  TipoPuntNodo cursor, anterior, aux;

  encontrado = false;
  cursor = secuencia;
  anterior = NULL;
  while (cursor != NULL) {
    aux = cursor->siguiente;
    if (cursor->valor == numero) {  /* encontrado, borrarlo */
      encontrado = true;
      if (anterior != NULL) {
        anterior->siguiente = cursor->siguiente;
      } else {
        secuencia = cursor->siguiente;
      }
      delete cursor;
    } else {    /* no encontrado */
      anterior = cursor;
    }
    cursor = aux;
  }
}

/** Imprimir una secuencia de números */
void Imprimir( TipoSecuencia secuencia ) {
  TipoPuntNodo cursor;

  cursor = secuencia;
  while (cursor != NULL) {
    printf( "%5d", cursor->valor );
    cursor = cursor->siguiente;
  }
  printf( "\n" );
}
```

```
int main() {
  int numero;
  TipoSecuencia secuencia;
  bool borrado;

  /*-- Leer los datos y almacenarlos --*/
  printf( "Datos leídos:\n" );
  secuencia = NULL;
  scanf( "%d", &numero );   /* primer dato */
  while (numero != 0) {
    printf( " %d", numero );
    InsertarEnOrden( secuencia, numero );
    scanf( "%d", &numero );   /* siguiente dato */
  }
  printf( "\n" );

  /*-- Mostrar la lista ya ordenada --*/
  printf( "\nDatos ordenados\n" );
  Imprimir( secuencia );

  /*-- Bucle de búsqueda y borrado de números --*/
  printf( "\nDatos a borrar:\n" );
  scanf( "%d", &numero );   /* primer dato */
  while (numero != 0) {
    Eliminar( secuencia, numero, borrado );
    if ( borrado) {
      printf( " %d borrado\n", numero );
      Imprimir( secuencia );
    } else {
      printf( " %d no encontrado\n", numero );
    }
    scanf( "%d", &numero );   /* siguiente dato */
  }
}
```

El programa funcionará igual que antes. Por lo tanto se reproducirá el ejemplo de funcionamiento con los mismos datos y resultados que se mostraron antes.

## 13.6 Punteros y vectores en C y C++

En C/C++ existe una estrecha relación entre las formaciones y los punteros. Sin embargo desde un punto de vista metodológico esta relación resulta bastante confusa, disminuye la claridad y aumenta la ambigüedad de los programas. Por esta razón, en este libro y en el lenguaje **C±** se ha tratado de

separar ambos conceptos de manera expresa. Lamentablemente, en el lenguaje **C±** no ha sido posible incorporar restricciones sintácticas capaces de impedir el manejo de punteros como formaciones debido al carácter semántico de esta relación. Por ello, es en el *Manual de Estilo* donde se incorpora una regla de obligado cumplimiento que prohíbe el uso de punteros como formaciones.

En las siguientes secciones se explica la analogía que existe en C/C++ entre los punteros y las formaciones, mostrando algunas de las posibilidades y consecuencias de este hecho. La razón de estas explicaciones es aclarar algunas de las irregularidades que presenta el lenguaje **C±** en el manejo de las formaciones. Por tanto, en estas siguientes secciones se muestran operaciones que están expresamente prohibidas y que no se deben utilizar en los programas.

### 13.6.1 Nombres de vectores como punteros

Cuando se ha declarado una variable de tipo vector el nombre de dicha variable equivale en gran medida a un puntero que apunta al comienzo del vector. El siguiente fragmento de código es perfectamente válido en C/C++:

```
typedef int* Tp_Entero;

typedef int Tv_Entero[5];

void Incrementar( Tp_Entero val ) {  /* paso por referencia */
  *val = *val + 1;
}

void ImprimirVector( const int v[], int nV ) {
  for (int k=0; k<nV; k++) {
    printf( "%10d", v[k] );
  }
  printf( "\n" );
}

Tp_Entero p1;
Tv_Entero v1 = {10, 20, 30, 40, 50};

p1 = v1;                        /* vector como puntero */

Incrementar( p1 );
ImprimirVector( p1, 5 );        /* puntero como vector */

Incrementar( v1 );              /* vector como puntero */
ImprimirVector( v1, 5 );
```

```
Incrementar( &v1[0] );
ImprimirVector( &v1[0], 5 );  /* puntero como vector */
```

El nombre del vector `v1` equivale al valor de un puntero que señala al comienzo, es decir, al primer elemento `v1[0]`. Por lo tanto puede ser asignado a una variable puntero del tipo equivalente. En sentido contrario, un puntero de ese tipo puede ser usado como argumento al invocar un subprograma que requiera un vector. Tras la asignación `p1 = v1` en el ejemplo, cada una de las parejas de sentencias `Incrementar(); ImprimirVector();` realizan la misma acción: incrementar el valor del primer elemento y luego imprimir los 5 elementos del vector.

La analogía alcanza incluso a poder utilizar el puntero como base para indexar un elemento del vector. Tras ejecutar `p1 = v1` las expresiones siguientes son equivalentes:

`p1[3]` equivale a `v1[3]`

Y aún más, en la declaración de un puntero se puede crear e inicializar un vector al que apunte:

```
typedef char * TipoPalabra;

TipoPalabra nombre = "Juan Antonio";
```

A pesar de lo anterior una variable puntero es claramente distinta de una variable vector. La asignación de un puntero a un vector

```
v1 = p1;
```

no es aceptada por el compilador y provoca un mensaje de error.

### 13.6.2 Paso de vectores como punteros

Tras las explicaciones anteriores debe quedar claro por qué el paso de vectores como argumentos se trata de manera diferente al paso como argumento de otros tipos de valores. Repetimos aquí algunos fragmentos de código del tema 9:

```
const int NumeroElementos = 10;
typedef int TipoVector[NumeroElementos];

void LeerVector( TipoVector v ) {...}          /* paso por referencia */

void EscribirVector( const TipoVector v ) {...}  /* paso por valor */
```

Las cabeceras de los subprogramas son en realidad equivalentes a las siguientes:

```
typedef int* TipoPuntero;

void LeerVector( TipoPuntero pv ) {...}       /* paso por referencia */

void EscribirVector( const TipoPuntero pv ) {...} /* paso por valor */
```

Tal como se ha dicho antes en este tema, el paso por valor de un puntero equivale al paso por referencia de la variable apuntada. El cualificador `const` delante de un argumento formal de tipo puntero indica al compilador que ese valor de tipo puntero no debe ser usado para modificar la variable a la que apunta.

### 13.6.3 Matrices y vectores de punteros

Si un puntero es análogo a un vector, entonces un vector de punteros es análogo a una matriz, es decir, a un vector de vectores. Veamos primero un ejemplo de declaración y uso de una matriz de enteros con 10 filas de 15 elementos:

```
typedef int TipoFila[15];
typedef TipoFila TipoMatriz[10];

TipoMatriz matriz;

matriz[3][5] = 27;
```

Y ahora un ejemplo similar usando un vector de punteros:

```
typedef int * TipoPuntero;
typedef TipoPuntero TipoMatriz[10];

TipoMatriz matriz;

matriz[3][5] = 27;
```

Este segundo ejemplo es sintácticamente correcto, pero sólo es realmente válido si los punteros señalan a vectores de enteros. Para que la analogía sea total es necesario crear dinámicamente cada fila de la matriz a partir de cada elemento del vector de punteros:

```
for (int k=0; k<=10; k++) {
  matriz[k] = new TipoFila;
}
```

La figura 13.9 muestra la diferencia entre una matriz propiamente dicha, es decir, un vector de filas, y su estructura análoga en forma de vector de punteros a filas. En ambos casos un elemento se designa como `matriz[fila][columna]`, pero la organización de los datos en memoria es claramente diferente.

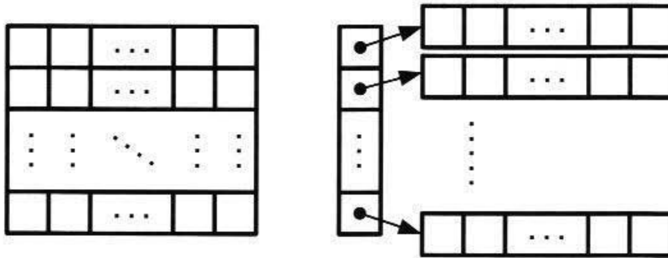


Figura 13.9 Matriz y vector de punteros a filas.

De hecho la segunda alternativa no exige que todas las filas tengan el mismo tamaño. Esto permite, por ejemplo, construir "diccionarios" en forma de vectores de palabras que son a su vez vectores de caracteres, pero cada una de un tamaño diferente:

```
typedef char * TipoPalabra;
typedef TipoPalabra TipoDiccionario[7];

TipoDiccionario nombreDias = {
  "Lunes",
  "Martes",
  "Miércoles",
  "Jueves",
  "Viernes",
  "Sábado",
  "Domingo"
};
```

En el lenguaje C/C++ el manejo de punteros permite realizar otras estructuras y operaciones más complejas y sofisticadas pero que habitualmente dan lugar a programas enrevesados y muy difíciles de comprender. Dichas operaciones, además de quedar fuera del alcance de esta asignatura, también estarían expresamente prohibidas en **C±**.

# Tema 14

# Tipos abstractos de datos

En este tema se da una introducción a los tipos abstractos de datos (TADs), como adelanto de lo que es la programación orientada a objetos, que excede del ámbito de este libro. Se introduce el concepto de tipo abstracto y las técnicas para programarlos como tipos registro (`struct`) en **C±**.

Aunque los tipos `struct` de C++ son en realidad clases, en **C±** se plantean de manera algo restringida, como un mecanismo relativamente sencillo para la programación de tipos abstractos.

## 14.1 Concepto de tipo abstracto de datos (TAD)

En programación tradicional, se identificaba el concepto de tipo de dato con el del conjunto de valores que pueden tomar los datos de ese tipo. De hecho esta idea se ha aplicado más a la forma de representación de los valores que a los valores en sí. Por ejemplo, si tenemos que operar con valores correspondientes a los meses del año, lo podremos hacer representándolos mediante números (1 al 12) o bien mediante cadenas de caracteres ('ENE', 'FEB', etc.), entre otras posibilidades. Estas representaciones se asocian con diferentes tipos de datos. En **C±** escribiríamos:

```
typedef int TipoMes;
```

o bien:

```
typedef char TipoMes[4];  /* 3 caracteres + carácter nulo al final */
```

Un enfoque más moderno de la programación trata de asociar la idea de tipo de datos con la clase de valores, abstractos, que pueden tomar los datos. Esto

quiere decir que la representación o codificación particular de los valores no cambia, de hecho, el tipo del dato considerado. Un paso adelante en este sentido ha sido la introducción de los tipos enumerados, en los que se definen colecciones de valores, abstractos, asociados a identificadores utilizables dentro del programa.

Como ya se ha dicho, los valores de los tipos enumerados no son valores numéricos, ni cadenas de caracteres, aunque pueden transformarse en esas otras formas de representación usando apropiadamente los mecanismos del lenguaje. Por ejemplo:

```
typedef enum TipoMes { Enero, Febrero, ... Diciembre };
TipoMes mes;

printf( "%d", int(mes)+1 );
```

En el enfoque actual de la programación se identifican los tipos de datos de forma completamente abstracta, llegando a la idea de *tipo abstracto de datos* (TAD). Esto quiere decir que un programa que use ese tipo de datos no debería necesitar ningún cambio por el hecho de modificar la representación o codificación de los valores de ese tipo. Si analizamos con cuidado qué necesita un programa para poder usar datos de un tipo, encontraremos que hace falta:

- Hacer referencia al tipo en sí, mediante un nombre, para poder definir variables, subprogramas, etc.
- Hacer referencia a algunos valores particulares, generalmente como constantes con nombre.
- Invocar operaciones de manipulación de los valores de ese tipo, bien usando operadores en expresiones aritméticas o bien mediante subprogramas.

El conjunto de todos estos elementos constituye el tipo abstracto de datos (TAD):

Un *tipo abstracto de datos* (TAD) es una agrupación de una *colección de valores* y una *colección de operaciones* de manipulación.

Es importante comprender que estas colecciones son cerradas, es decir sólo se deben poder usar los valores abstractos y las operaciones declaradas para ese tipo. Además los detalles de cómo se representan los valores y cómo se implementan las operaciones pueden estar ocultos para quien utiliza el tipo abstracto. Esto no ocurría cuando se asociaba el tipo de valor con su forma de representación. Por ejemplo, si representamos los meses del año mediante números, podremos usar el número 33, aunque no sea ningún mes válido, al

igual que podremos multiplicar los números de dos meses, aunque esto no tenga ningún sentido.

La programación *orientada a objetos*, ampliamente usada en la actualidad, se basa esencialmente en el uso de tipos abstractos de datos, con algunas modificaciones. La terminología cambia bastante: se habla de *clases* y *objetos* en lugar de tipos y datos, y de *métodos* en lugar de operaciones. Por otra parte se añade el concepto de *herencia* para facilitar la definición de nuevas clases a partir de otras ya existentes, reutilizando elementos ya definidos.

Este tema se limita a introducir el concepto de tipo abstracto de datos, que es perfectamente suficiente para desarrollar una buena metodología de desarrollo de programas. La programación orientada a objetos se deja para un estudio posterior.

## 14.2 Realización de tipos abstractos en **C±**

Cuando se utiliza un lenguaje de programación orientado a objetos el mecanismo de clases del que dispone es perfectamente adecuado para programar tipos abstractos de datos, ya que de hecho las clases son implícitamente TADs. Eso ocurre, por ejemplo con C++.

Sin embargo, el subconjunto denominado **C±** que se usa en este libro no contempla la definición de clases como tales, sino que se limita a presentar una forma algo más limitada de programar tipos abstractos de datos. Para ello se aprovecha el hecho de que los tipos registro (`struct`) se tratan en C++ como equivalentes a clases.

### 14.2.1 Definición de tipos abstractos como tipos registro (`struct`)

Hasta ahora se ha visto que los tipos registro permiten definir estructuras con varios campos de datos con nombre y tipo individual. Ahora añadiremos la posibilidad de incluir también otros elementos, en particular subprogramas, y distinguir entre elementos públicos y privados. De esta manera se pueden definir tipos abstractos de datos, ya que:

- Los campos de datos sirven para almacenar el contenido de información del dato abstracto.
- Los subprogramas permiten definir operaciones sobre esos datos.

- La posibilidad de declarar ciertos elementos como privados permite ocultar detalles de implementación, y dejar visible sólo la interfaz del tipo abstracto.

Para ello las reglas de sintaxis de la declaración de un tipo registro se amplían de la siguiente forma:

*Tipo_struct* **::=** typedef struct *Identificador*
  { *Lista_de_items* **[** private: *Lista_de_items* **]** } ;

*Lista_de_items* **::=** *Item* ; **{** *Item* ; **}**

*Item* **::=** *Campo* **|** *Cabecera_subprograma*

*Campo* **::=** *Campos_igual_tipo* **|** *Campo_puntero* **|** *Campo_array*

*Campos_igual_tipo* **::=** *Identificador_de_tipo Lista_de_identificadores*

*Campo_puntero* **::=** *Identificador_de_tipo * Identificador*

*Campo_array* **::=** *Identificador_de_tipo Identificador Dimensiones*

Como ejemplo se muestra una declaración del tipo abstracto `TipoPunto`, correspondiente a un punto en el plano euclídeo. Cada punto se representa por sus coordenadas cartesianas, y se le asocian subprogramas para leer y escribir puntos (sus coordenadas), y para calcular la distancia entre dos puntos.

```
typedef struct TipoPunto {
  float x;  /* Coordenada X */
  float y;  /* Coordenada Y */

  /** Leer un punto con formato "(x,y)" */
  void Leer();
  /** Escribir un punto con formato "(x,y)" */
  void Escribir();
  /** Calcular la distancia de un punto a otro */
  float Distancia( TipoPunto p );
};
```

Como puede verse, los subprogramas correspondientes a las operaciones sobre el tipo abstracto se declaran simplemente por su cabecera, porque lo que se declara aquí es sólo la interfaz del tipo. La notación para referirse a las operaciones es la misma que para los campos de datos, usando el punto (`.`) como operador de cualificación. El esquema de esta notación y un ejemplo de uso son:

| | |
|---|---|
| *variable*`.`*campo* | Referencia a campo de datos |
| *variable*`.`*operación*`(` *argumentos* `)` | Referencia a operación |

```
TipoPunto p, q;

p.x = 3.3;
p.y = 4.4;

p.Leer();
p.Escribir();
printf( "%f", p.Distancia( q ) );
```

Como complemento de la declaración de la interfaz se necesita además definir la implementación de las operaciones. Esta implementación se hace fuera de la declaración del tipo registro, usando la notación *Tipo*::*Operación* como nombre del subprograma. El código de implementación sería:

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

/* Excepción en lectura */
typedef enum TipoPuntoError { PuntoNoLeido };

/** Leer un punto con formato "(x,y)" */
void TipoPunto::Leer() {
  int campos;

  campos = scanf( " ( %f , %f )", &x, &y );
  if (campos < 2) {  /* comprobar que se han leído dos valores */
    throw PuntoNoLeido;
  }
}

/** Escribir un punto con formato "(x,y)" */
void TipoPunto::Escribir() {
  printf( "(%f, %f)", x, y );
}

/** Calcular la distancia de un punto a otro */
float TipoPunto::Distancia( TipoPunto p ) {
  float deltaX, deltaY;

  deltaX = x - p.x;
  deltaY = y - p.y;
  return sqrt( deltaX*deltaX + deltaY*deltaY );
}
```

Como se ve, en el código de implementación se puede hacer referencia a los campos de la estructura directamente por su nombre, sin necesidad de cuali-

ficación. Se sobreentiende que se refieren a la propia variable sobre la que se invoca la operación.

Adicionalmente en este ejemplo se ha utilizado el subprograma **scanf** como función, y no como procedimiento (en C y C++ no hay diferencia entre ambos tipos de subprograma). Se aprovecha el resultado de la llamada, que es el número de valores efectivamente leídos, para comprobar si hay errores en la lectura o se alcanza el final de los datos.

▮ NOTA: Las operaciones definidas en los módulos de librería estándar de C quedan fuera del lenguaje **C±** en sí. Por lo tanto esas operaciones estándar podrán usarse con la misma libertad que en lenguaje C.

Una vez definido el tipo abstracto se puede escribir código que lo use. Por ejemplo, se puede escribir un programa que lea segmentos definidos por sus puntos extremos, y calcule e imprima sus longitudes:

```
#include <stdio.h>

/* Definición del tipo abstracto PUNTO */
/* ... aquí se incluye el código anterior ... */

/** Programa principal */
int main() {
  TipoPunto a, b;
  bool seguir = true;

  while (seguir) {
    try {
      a.Leer();
      b.Leer();
      printf( "Segmento: " );
      a.Escribir();
      printf( " " );
      b. Escribir();
      printf( " Longitud: %f\n", a.Distancia( b ) );
    } catch (TipoPuntoError e) {
      seguir = false;
    }
  }
}
```

El programa se plantea como un bucle indefinido que va leyendo pares de puntos y calcula e imprime la longitud del segmento que definen. El programa termina cuando se produce una excepción en la lectura de un punto, bien porque se termina el fichero de datos de entrada o porque aparece un dato que

no es una representación válida de un punto ($x$, $y$), con los paréntesis y la coma de separación.

Combinando los fragmentos de código anteriores podemos mostrar ya el programa completo:

```
/************************************************
* Programa: Segmentos
*
*   Este programa lee una serie de segmentos, dados
*   por sus extremos, y calcula sus longitudes.
************************************************/
#include <stdio.h>
#include <math.h>

/*================================================
    Interfaz del tipo abstracto PUNTO
================================================*/
typedef struct TipoPunto {
  float x;  /* Coordenada X */
  float y;  /* Coordenada Y */

  /* Leer un punto con formato "(x,y)" */
  void Leer();
  /* Escribir un punto con formato "(x,y)" */
  void Escribir();
  /* Calcular la distancia de un punto a otro */
  float Distancia( TipoPunto p );
};

/* Excepción en lectura */
typedef enum TipoPuntoError { PuntoNoLeido };

/*================================================
    Implementación del tipo abstracto PUNTO
================================================*/
/** Leer un punto con formato "(x,y)" */
void TipoPunto::Leer() {
  int campos;

  campos = scanf( " ( %g , %g )", &x, &y );
  if (campos < 2) {
    throw PuntoNoLeido;
  }
}
```

```
/** Escribir un punto con formato "(x,y)" */
void TipoPunto::Escribir() {
  printf( "(%g, %g)", x, y );
}

/** Calcular la distancia de un punto a otro */
float TipoPunto::Distancia( TipoPunto p ) {
  float deltaX, deltaY;

  deltaX = x - p.x;
  deltaY = y - p.y;
  return sqrt( deltaX*deltaX + deltaY*deltaY );
}

/*=====================================================
    Programa principal
=====================================================*/
int main() {
  TipoPunto a, b;
  bool seguir = true;

  while (seguir) {
    try {
      a.Leer();
      b.Leer();
      printf( "Segmento: " );
      a.Escribir();
      printf( " " );
      b. Escribir();
      printf( " Longitud: %g\n", a.Distancia( b ) );
    } catch (TipoPuntoError e) {
      seguir = false;
    }
  }
}
```

Ejemplo de datos de entrada:

```
(0, 4) (3, 0)
   (1,2)(3,4)


  (5,
0) (   6     ,      0    )

(-1, 1)  (1, 1)
No hay más puntos
```

Resultados obtenidos:

```
Segmento: (0, 4) (3, 0) Longitud: 5
Segmento: (1, 2) (3, 4) Longitud: 2.82843
Segmento: (5, 0) (6, 0) Longitud: 1
Segmento: (-1, 1) (1, 1) Longitud: 2
```

### 14.2.2 Ocultación

Para que un tipo sea realmente abstracto haría falta que los detalles de implementación no fueran visibles. Los subprogramas, como mecanismo de *abstracción*, ya ocultan los detalles de la realización de las operaciones. Sin embargo queda la cuestión de cómo ocultar la manera de representar los valores del tipo abstracto.

En el ejemplo anterior al definir el `TipoPunto` se tienen visibles los campos de información de sus coordenadas, que pueden ser consultadas y modificadas por el código que usa el tipo. En este caso no se presenta ningún problema por eso, ya que cualquier pareja de coordenadas define un punto válido.

No ocurre lo mismo con otros tipos de datos. Por ejemplo, si se almacena el valor de una fecha como la tupla numérica (*día*, *mes*, *año*) no se puede admitir cualquier combinación de valores. La fecha (20, 7, 2009) es correcta, pero (7, 20, 2009) no lo es, y menos aún (50, -3, 54321). Si se quiere definir el tipo fecha como tipo abstracto será necesario ocultar los detalles de representación interna de los valores, de manera que sólo se puedan construir fechas usando operaciones que garanticen la corrección de los valores del tipo.

Para permitir esta ocultación los tipos `struct` admiten la posibilidad de declarar ciertos elementos componentes como privados, usando la palabra clave `private` para delimitar una zona de declaraciones privadas dentro de la estructura. La interfaz básica del `TipoFecha` podría ser:

```
typedef struct TipoFecha {
  /* Dar valor a un dato fecha */
  void Poner( int dia, int mes, int anno );

  /* Obtener el contenido de un dato fecha */
  int Dia();
  int Mes();
  int Anno();

private:
  int dia, mes, anno;
};
```

Como contrapartida a ocultar los elementos internos de representación de las fechas, ha sido necesario añadir operaciones explícitas para asignar valor y recuperar el contenido de una fecha. Estas operaciones son las más básicas. Las aplicaciones informáticas que operan con datos de tipo fecha son enormemente frecuentes. En ellas se realizan operaciones muy diversas, de manera que la interfaz de un tipo fecha de uso general debería incluir decenas de operaciones. Citaremos sólo algunas posibilidades, como ejemplo:

```
typedef struct TipoFecha {

  void Leer();
  void Escribir( const char formato[] );

  int DiasHasta( TipoFecha f );
  void Incrementar( int dias );

  bool EsAnterior( TipoFecha f );
  bool EsIgual( TipoFecha f );
  bool EsCorrecta();

  TipoDiaSemana DiaSemana();
  ... etc. ...
};
```

### 14.2.3 Ejemplo: Imprimir fechas en orden

Como ejemplo se presenta aquí una nueva versión del programa de leer fechas e imprimirlas en orden, que ya apareció en el tema 12. Ahora se define el tipo fecha como tipo abstracto de datos y se simplifica la lectura del mes para no hacer el código demasiado largo.

```
/*************************************************
* Programa: Fechas2
*
* Descripción:
*   Programa que lee una serie de fechas,
*   comprueba que son correctas, y las
*   imprime en orden cronológico.
*************************************************/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
```

```
/*===========================================
    Tipo abstracto FECHA
  ===========================================*/
typedef struct TipoFecha {

  bool Leer();
  void Escribir();
  bool EsCorrecta();
  bool EsPosterior( TipoFecha f );

private:
  int dia, mes, anno;
};

/*===========================================
    Lista de fechas
  ===========================================*/
const int maxFechas = 100;
typedef TipoFecha listaFechas_t[maxFechas];

listaFechas_t lista;          /* lista de fechas leídas, en orden */
int numFechas;                /* número de fechas leídas */

/*===========================================
    Nombres de los meses
  ===========================================*/
const int maxNombre = 15;
typedef char TipoNombreMes[maxNombre];
typedef TipoNombreMes listaNombres_t[13];

listaNombres_t nombres  = {"?",        /* sin nombre cuando mes = 0 */
                           "Enero",    /* nombres de los meses */
                           "Febrero",
                           "Marzo",
                           "Abril",
                           "Mayo",
                           "Junio",
                           "Julio",
                           "Agosto",
                           "Septiembre",
                           "Octubre",
                           "Noviembre",
                           "Diciembre"
                          };
```

```
/*=========================================
    Implementación de operaciones con FECHAs
  =========================================*/

/** Comprobar fecha correcta */
bool TipoFecha::EsCorrecta() {

  if ( dia<=0 || mes<=0 || anno<=0) {
    return false;
  }
  switch (mes) {
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
    return (dia <= 31);
    break;
  case 4:
  case 6:
  case 9:
  case 11:
    return (dia <= 30);
    break;
  case 2:
    if (anno%4 == 0 && anno%100 != 0 ||
        anno%400 == 0) {
      return (dia <= 29);
    } else {
      return (dia <= 28);
    };
    break;
  default:
    return false;
  }
}

/** Escribir fecha día-mes-año */
void TipoFecha::Escribir() {
  printf( "%2d-%s-%4d", dia, nombres[mes], anno );
}

/** Leer mes en número o en letra completo o abreviado */
void LeerMes( int & mes ) {
```

```
  TipoNombreMes nombre;
  bool encontrado;

  scanf( "%s", nombre );          /* leer mes como texto */
  if (isdigit(nombre[0])) {       /* mes en número */
    mes = atoi( nombre );
  } else {                        /* mes en letra */
    mes = 13;
    encontrado = false;
    while (mes > 0 && !encontrado) {
      mes--;
      encontrado =                /* coinciden 3 caracteres */
        nombres[mes][0] == toupper(nombre[0]) &&
        nombres[mes][1] == tolower(nombre[1]) &&
        nombres[mes][2] == tolower(nombre[2]);
    }
  }
}

/** Leer fecha "dia mes año" con el mes en número
   o letra y el año con dos o cuatro cifras.
   Devuelve 'true' si ha podido leer */
bool TipoFecha::Leer() {
  dia = 0;
  scanf( "%d", &dia );
  if (dia == 0) {
    return false;
  }
  LeerMes( mes );
  scanf( "%d", &anno );
  if (anno < 100) {
    anno = anno + 2000;
  }
  return true;
}

/** Comparar dos fechas */
bool TipoFecha::EsPosterior( TipoFecha f ) {
  if (anno != f.anno) {
    return (anno > f.anno);
  }
  if (mes != f.mes) {
    return (mes > f.mes);
  }
  return (dia > f.dia);
}
```

```
/** Insertar una fecha en la lista ordenada */
void InsertarFecha( TipoFecha f ) {
  int izquierda, derecha, centro;

  /*-- Poner la fecha como centinela, al final (evita
    que añadir al final sea un caso especial) --*/
  lista[numFechas] = f;
  numFechas++;

  /*-- Determinar la posición que le corresponde --*/
  izquierda = 0;
  derecha = numFechas-1;
  while (izquierda < derecha) {
    centro = (izquierda+derecha)/2;
    if (f.EsPosterior( lista[centro] )) {
      izquierda = centro+1;
    } else {
      derecha = centro;
    }
  }

  /*-- Hacer sitio para la nueva fecha --*/
  for (int ind = numFechas-1; ind >= izquierda; ind--) {
    lista[ind+1] = lista[ind];
  }

  /*-- Colocar la fecha en su sitio --*/
  lista[izquierda] = f;
}

/*============================================
    Programa principal
  ============================================*/
int main() {
  TipoFecha fecha;
  bool seguir;

  printf( "Fechas leídas:\n" );
  numFechas = 0;
  seguir = fecha.Leer();
  while (seguir && numFechas < maxFechas) {
    fecha.Escribir();
    if (fecha.EsCorrecta()) {
      InsertarFecha( fecha );
    } else {
```

```
      printf( " ** incorrecta **" );
    }
    printf( "\n" );
    seguir = fecha.Leer();
  }

  printf( "\nFechas en orden:\n" );
  for (int k=0; k<numFechas; k++) {
    lista[k].Escribir();
    printf( "\n" );
  }
}
```

Ejemplo de datos de entrada:

```
10 Marzo 1972
30 feb 82
29 FEB 1900
11 3 72
29 FEB 2000
28 diciem 1993
4 enero 91
15 error 89
10 10 10
0
```

Resultados obtenidos:

```
Fechas leídas:
10-Marzo-1972
30-Febrero-2082 ** incorrecta **
29-Febrero-1900 ** incorrecta **
11-Marzo-2072
29-Febrero-2000
28-Diciembre-1993
 4-Enero-2091
15-?-2089 ** incorrecta **
10-Octubre-2010

Fechas en orden:
10-Marzo-1972
28-Diciembre-1993
29-Febrero-2000
10-Octubre-2010
11-Marzo-2072
 4-Enero-2091
```

## 14.3 Metodología basada en abstracciones

La técnica de programación estructurada, basada en refinamientos sucesivos, puede ampliarse para contemplar la descomposición modular de un programa. La metodología de desarrollo será esencialmente la misma que se ha presentado en el tema 8, referente al desarrollo usando abstracciones en forma de subprogramas (*abstracciones funcionales*). La diferencia es que ahora disponemos también de un nuevo mecanismo de abstracción, que son los *tipos abstractos de datos*.

Igualmente son de aplicación las técnicas generales de desarrollo: descendente o ascendente, que pueden plantearse no sólo con abstracciones funcionales sino también con abstracciones de datos. En cualquier caso el desarrollo deberá atender tanto a la organización de las operaciones como a la de los datos sobre las que operan, de manera que habrá que ir realizando simultáneamente las siguientes actividades:

- Identificar las operaciones a realizar, y refinarlas.
- Identificar las estructuras de información, y refinarlas.

Como se verá, puede establecerse una analogía entre ambas.

### 14.3.1 Desarrollo por refinamiento basado en abstracciones

Comenzaremos por recordar lo que se decía en tema 8 sobre el *desarrollo descendente* con abstracciones funcionales: en cada etapa de refinamiento de una operación hay que optar por una de las alternativas siguientes:

- Considerar la operación como *operación terminal*, y codificarla mediante sentencias del lenguaje de programación.
- Considerar la operación como *operación compleja*, y descomponerla en otras más sencillas.
- Considerar la operación como *operación abstracta*, y especificarla, escribiendo más adelante el subprograma que la realiza.

Ahora podemos reformular estas opciones para las estructuras de datos a utilizar:

- Considerar el dato como un *dato elemental*, y usar directamente un tipo predefinido del lenguaje para representarlo.
- Considerar el dato como un *dato complejo*, y descomponerlo en otros más sencillos (como registro, unión o formación).
- Considerar el dato como un *dato abstracto* y especificar su interfaz, dejando para más adelante los detalles de su implementación.

Aplicaremos estas técnicas de refinamiento a un programa que construya y dibuje de manera aproximada una de las llamadas Curvas-C. Estas curvas, que son realmente líneas poligonales realizables sobre una cuadrícula, se definen *recursivamente*. Hay toda una familia de Curvas-C, que se designan mediante un número de orden, empezando por cero:

- C(0) es un trazo recto de longitud unidad.
- C(n), para n>0, equivale a dibujar dos veces C(n-1), con un giro de 90° a la derecha entre ambas.

Las primeras curvas de la familia se muestran en la figura 14.1.

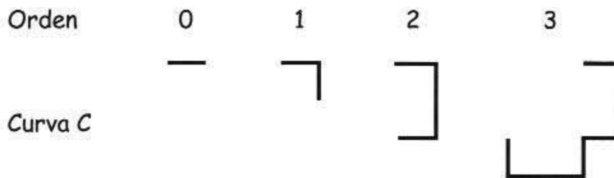



Figura 14.1 Familia de Curvas-C.

Para resolver el problema tendremos en cuenta que el procedimiento `printf` disponible para escribir resultados exige ir imprimiendo las líneas una a una, de arriba a abajo. Por lo tanto es preciso componer el dibujo completo antes de imprimirlo. La acción principal se descompone inicialmente en una secuencia de acciones sencillas, tal como:

*Imprimir Curva C* ⟶
 *Iniciar la página en blanco*
 *Componer el dibujo de la curva*
 *Imprimir la página*

*Componer el dibujo de la curva* ⟶

```
if (el orden es 0) {
  Trazar un segmento unidad
} else if (el orden es > 0) {
  Componer la curva de orden anterior
  Girar a la derecha
  Componer la curva de orden anterior
  Girar a la izquierda
}
```

Hasta aquí la descomposición funcional inicial. Ahora toca identificar las estructuras de datos necesarias para mantener la información necesaria y poder implementar las operaciones. La descripción anterior ya contiene implícitamente algunas decisiones sobre cómo almacenar y componer el dibujo. En

concreto se está asumiendo la técnica de *gráficos de tortuga* empleada en el lenguaje LOGO. Esta técnica consiste en disponer de un agente capaz de ir moviéndose sobre el plano al tiempo que deja un trazo o rastro de su movimiento. A este agente le llamaremos "tortuga", y se le puede ordenar que avance y que gire.

En cuanto a la posibilidad de componer el dibujo completo antes de imprimirlo, se asume que la tortuga se mueve sobre un "papel" que almacena los trazos y hace una función de memoria o registro gráfico. Corresponde a lo que en los refinamientos anteriores se ha denominado *página*.

Pasaremos ahora a identificar las acciones a realizar con la tortuga y el papel. De los refinamientos anteriores podemos deducir que se necesitarán al menos las operaciones siguientes:

*Para la tortuga*
- *Avanzar la tortuga un paso*
- *Girar la tortuga a la derecha*
- *Girar la tortuga a la izquierda*

*Para el papel*
- *Iniciar el papel en blanco*
- *Registrar un trazo horizontal*
- *Registrar un trazo vertical*
- *Imprimir el contenido del papel*

Estas operaciones se realizan, en general, sobre el tipo de dato al que corresponden, excepto la operación de avanzar la tortuga, que ha de registrar al mismo tiempo el trazo sobre el papel, bien sea horizontal o vertical, dependiendo de su orientación en ese momento.

Ahora ya se dispone de una idea clara de la organización del programa, que usaría las siguientes abstracciones:

- Un subprograma para generar *recursivamente* la Curva-C de un orden dado.
- Un tipo abstracto de datos que implemente el agente de dibujo, es decir, la tortuga.
- Un tipo abstracto de datos que implemente el registro del dibujo, es decir, el papel.

Con un poco de experiencia se pueden prever las operaciones complementarias y los datos internos que serán necesarios para desarrollar el programa completo. En este caso pasaremos a presentar sin más unas posibles interfaces de las abstracciones reconocidas hasta el momento:

```
/*************************************************
* Procedimiento: CurvaC
*************************************************/

void CurvaC( int orden, TipoTortuga & t, TipoPapel & p );

/*************************************************
* Tipo abstracto: Papel
*************************************************/
typedef struct TipoPapel {
  void PonerEnBlanco();
  void MarcarHorizontal( int x, int y );
  void MarcarVertical( int x, int y );
  void Imprimir();
private:
  /* matriz con los trazos horizontales y verticales */
};

/*************************************************
* Tipo abstracto: Tortuga
*************************************************/
typedef enum TipoRumbo { Este, Norte, Oeste, Sur };

typedef struct TipoTortuga {
  void Poner( int x, int y, TipoRumbo rumbo );
  void Avanzar( TipoPapel & p );
  void GirarDerecha();
  void GirarIzquierda();
private:
  /* posición y orientación de la tortuga */
};
```

## 14.4 Ejemplo: Dibujar una *Curva-C*

Aquí se presenta el programa completo correspondiente al ejemplo introducido en la sección anterior. El programa responde a la descomposición obtenida por refinamientos sucesivos, y en él se han completado las interfaces de los módulos con los elementos auxiliares necesarios.

Tal como se apuntaba, se hace uso de tipos `struct` para definir la Tortuga y el Papel como tipos abstractos de datos. Sus estados se almacenan en estructuras de datos que se manejan internamente como datos privados.

La realización del tipo Papel limita arbitrariamente su tamaño a una cuadrícula de 19 líneas de 32 casillas. Cada casilla registra el trazo vertical en su borde izquierdo y el trazo horizontal en su borde inferior. Para imprimir el dibujo de manera aproximada cada casilla se imprime con dos caracteres. Con este tamaño el resultado del ejemplo puede incluirse fácilmente en el texto de este libro. Los trazos horizontales y verticales se aproximan con el guión bajo (_) y la barra vertical (|).

Según se muestra en la figura 14.2, el origen de coordenadas es el extremo inferior izquierdo. Los rumbos se miden en sentido antihorario, empezando por la orientación a la derecha, es decir, tal como se miden habitualmente los ángulos en la geometría del plano.

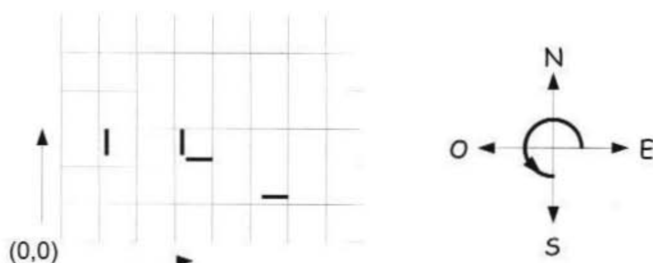



Figura 14.2 Composición del dibujo sobre una cuadrícula.

La posición inicial de la tortuga se ha establecido para conseguir que entre dentro del papel la Curva-C del orden más alto posible para el tamaño fijado. El programa es robusto en cuanto que:

- Ignora los intentos de dibujar curvas de orden negativo.
- Si se manda dibujar una curva que ocupa más espacio del disponible la composición de la curva se sigue haciendo correctamente pero los trazos fuera del papel no se registran.

El código desarrollado es, en general, bastante sencillo y puede ser leído sin grandes dificultades. El único detalle algo complicado es el cálculo de los rumbos, que exige convertir los valores enumerados a enteros en el rango 0-3, y viceversa. El listado completo es el siguiente:

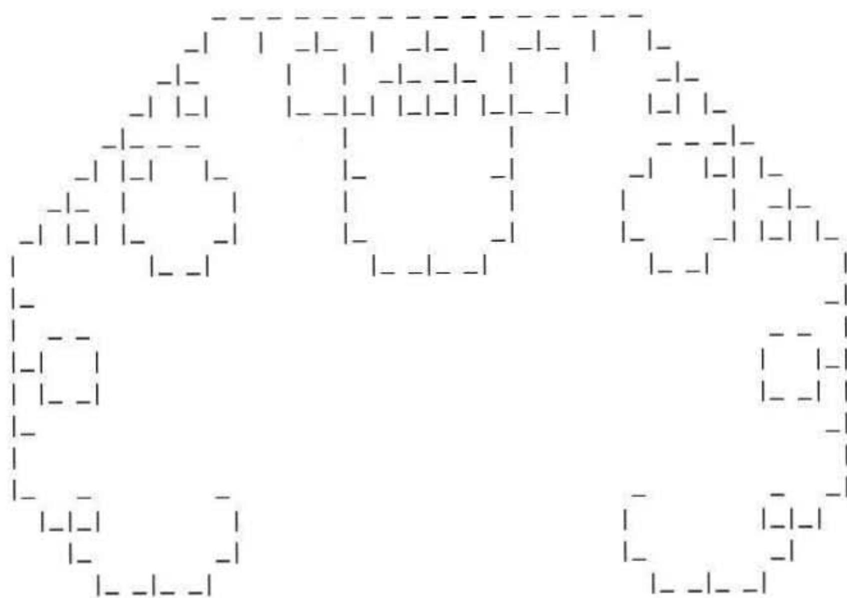
```
/*********************************************************************
* Programa: DibujarC
*
* Descripción:
*   Este programa lee como dato el orden de una curva C, y a
*   continuación la dibuja en forma de texto en pantalla
*********************************************************************/
```

```
#include <stdio.h>

/*================================================
    Parámetros globales
================================================*/
  const int ANCHO = 32;  /* cuadrícula en */
  const int ALTO = 19;   /* la pantalla */

/*================================================
    Tipo abstracto PAPEL
================================================*/
typedef struct TipoPapel {
  void PonerEnBlanco();
  void MarcarHorizontal( int x, int y );
  void MarcarVertical( int x, int y );
  void Imprimir();
private:
  bool Dentro( int x, int y );
  char marcasH[ANCHO][ALTO];
  char marcasV[ANCHO][ALTO];
};

bool TipoPapel::Dentro( int x, int y ) {
  return (x >= 0 && x < ANCHO && y >= 0 && y < ALTO);
}

void TipoPapel::PonerEnBlanco() {
  for (int x=0; x<ANCHO; x++) {
    for (int y=0; y<ALTO; y++) {
      marcasH[x][y] = ' ';
      marcasV[x][y] = ' ';
    }
  }
}

void TipoPapel::MarcarHorizontal( int x, int y ) {
  if (Dentro( x, y )) {
    marcasH[x][y] = '_';
  }
}

void TipoPapel::MarcarVertical(int x, int y ) {
  if (Dentro( x, y )) {
    marcasV[x][y] = '|';
  }
}
```

```
void TipoPapel::Imprimir() {
  for (int y=ALTO-1; y>=0; y--) {
    for (int x=0; x<ANCHO; x++) {
      printf( "%c%c", marcasV[x][y], marcasH[x][y]);
    }
    printf( "\n" );
  }
}

/*==============================================
    Tipo abstracto TORTUGA
==============================================*/
typedef enum TipoRumbo { Este, Norte, Oeste, Sur };

typedef struct TipoTortuga {
  void Poner( int x, int y, TipoRumbo rumbo );
  void Avanzar( TipoPapel & p );
  void GirarDerecha();
  void GirarIzquierda();
private:
  int xx, yy;
  TipoRumbo sentido;
};

void TipoTortuga::Poner( int x, int y, TipoRumbo rumbo ) {
  xx = x;
  yy = y;
  sentido = rumbo;
}

void TipoTortuga::Avanzar( TipoPapel & p ) {
  switch (sentido) {
  case Norte:
    p.MarcarVertical( xx, yy );
    yy++;
    break;
  case Sur:
    yy--;
    p.MarcarVertical( xx, yy );
    break;
  case Este:
    p.MarcarHorizontal( xx, yy );
    xx++;
    break;
  case Oeste:
```

```
    xx--;
    p.MarcarHorizontal( xx, yy );
    break;
  }
}

void TipoTortuga::GirarDerecha() {
  sentido = TipoRumbo( (int(sentido)-1+4) % 4 );
                      /* +4 evita rumbo negativo */
}

void TipoTortuga::GirarIzquierda() {
  sentido = TipoRumbo( (int(sentido)+1) % 4 );
}

/*============================================
    Función CURVA-C
============================================*/
void CurvaC( int orden, TipoTortuga & t, TipoPapel & p ) {
  if (orden == 0) {
    t.Avanzar( p );
  } else if (orden > 0) {
    CurvaC( orden - 1, t, p );
    t.GirarDerecha();
    CurvaC( orden - 1, t, p );
    t.GirarIzquierda();
  }
}

/*============================================
    Programa principal
============================================*/
int main() {
  int orden;
  TipoTortuga tt;
  TipoPapel pp;

  printf( "Orden: " );
  scanf( "%d", &orden );

  pp.PonerEnBlanco();
  /* posición inicial de conveniencia */
  tt.Poner( 8, 3, Este );
  CurvaC( orden, tt, pp );
  pp.Imprimir();
}
```

Ejemplo de ejecución del programa:



# Tema 15

# Módulos

En este tema se da una introducción a la programación modular, en especial basada en el empleo de tipos abstractos de datos.

Se introduce el concepto de módulo, en general, y su realización en **C±**, en particular. Se discuten brevemente las características necesarias para compilar módulos por separado, de forma segura, y la manera de conseguirlo en este lenguaje.

Finalmente se extiende la metodología de desarrollo de programas con la posibilidad de descomposición en módulos separados, estableciendo las recomendaciones del desarrollo modular basado en abstracciones.

## 15.1 Concepto de módulo

En programación, un *módulo* es, en general, un fragmento de programa utilizado en algún momento para la construcción del programa completo. Lo que distingue a un módulo propiamente dicho de un fragmento arbitrario del programa es el hecho de que en algún momento de la construcción haya sido reconocido como tal, y por tanto que se haya desarrollado o refinado de forma relativamente independiente del resto del programa. Podríamos definir:

*Módulo*: Fragmento de programa desarrollado de forma independiente.

El desarrollo independiente debe serlo en el máximo grado posible. Atendiendo a las técnicas de preparación de programas en lenguajes de programación simbólicos, diremos que un módulo debería ser compilado y probado por separado, y no tratarse de un simple fragmento de texto dentro de un único programa fuente.

La razón de exigir compilación por separado para los distintos módulos de un programa obedece a la necesidad de limitar la complejidad de aquello que está siendo elaborado por una persona en un momento dado. Si el módulo se va a compilar por separado, la persona que lo desarrolle podrá concentrarse en él, prescindiendo en parte de cómo se utiliza ese módulo desde el resto del programa. De la misma forma, quien escriba el resto del programa no se preocupará de los detalles de cómo está codificado el módulo, sino sólo de cómo hay que usarlo.

El concepto de módulo, por tanto, está íntimamente ligado a la idea de *abstracción*. Un módulo debe definir un elemento abstracto (o varios relacionados entre sí) y debe ser usado desde fuera con sólo saber *qué hace* el módulo, pero sin necesidad de conocer *cómo lo hace*.

### 15.1.1 Especificación y realización

Igual que en cualquier otro elemento abstracto, en un módulo podemos distinguir dos puntos de vista, correspondientes a su *especificación* y a su *realización*. Tal como se ha indicado con anterioridad, la primera visión nos dice qué hace el módulo, y la segunda nos dice cómo lo hace.

La especificación de un módulo que contenga la definición de una serie de elementos abstractos consistirá, fundamentalmente, en el conjunto de las especificaciones de cada uno de ellos, por separado, más una indicación de los posibles efectos de unos sobre otros cuando se usan de forma combinada.

La realización del módulo consistirá en la realización de cada uno de los elementos abstractos contenidos en dicho módulo.

La especificación del módulo es todo lo que se necesita para poder usar los elementos definidos en él. Esta especificación constituye la *interfaz* (en inglés *interface*) entre el módulo (incluida su realización) y el programa que lo usa.

La independencia entre la realización de un módulo y el programa que lo usa se incrementa si la realización de un elemento abstracto no es visible desde donde se usa. Esta característica se denomina *ocultación*, y ha sido ya desarrollada en los temas 7 y 14 para las funciones, procedimientos y tipos abstractos de datos.

Referida a los módulos, la ocultación consiste en que el programa que usa un elemento de un módulo sólo tiene visible la información de la interfaz, pero no la de la realización.

### 15.1.2 Compilación separada

Los lenguajes de programación que permiten programar usando módulos pueden emplear diversas técnicas para definirlos e invocar los elementos definidos en ellos. Tal como se ha comentado antes, es importante que los módulos puedan compilarse por separado. Esto quiere decir que los lenguajes de programación deben permitir escribir un programa complicado como un conjunto de varios ficheros fuente distintos, cada uno de los cuales pueda compilarse de manera más o menos independiente de los demás.

Por otra parte, para que el uso de los elementos de un módulo sea correcto, habrá que hacerlo de acuerdo con la interfaz establecida. La interfaz debe ser tenida en cuenta al compilar un programa que use elementos de un módulo separado. Por el contrario, la realización del módulo debe permanecer invisible para el programa que lo usa con objeto de mantener la deseable ocultación de los detalles de los elementos abstractos contenidos en él.

En la figura 15.1 se representa gráficamente la visibilidad deseable entre un módulo y el programa que lo usa.

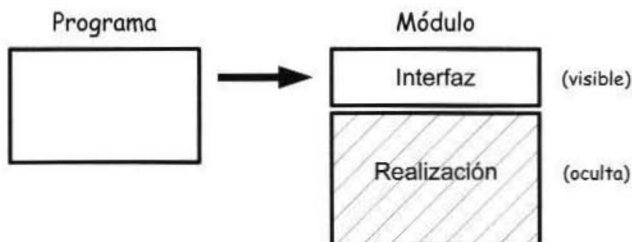



Figura 15.1 Visibilidad de un módulo.

Resumiendo:

*Compilación separada*: El programa está formado por varios ficheros fuente, cada uno de los cuales se compila por separado.

*Compilación segura*: Al compilar un fichero fuente el compilador comprueba que el uso de elementos de otros módulos es consistente con la interfaz.

*Ocultación*: Al compilar un fichero fuente el compilador no usa información de los detalles de realización de los elementos de otros módulos.

Entre las técnicas empleadas por lenguajes de programación de uso frecuente en lo que respecta a compilación separada, tenemos situaciones tales como las siguientes:

(a) El fichero del programa y del módulo se tratan de forma totalmente separada, sin visibilidad de la interfaz (lenguaje FORTRAN y las primeras versiones del lenguaje C).
(b) La parte necesaria de la interfaz se copia o importa manualmente en el programa que la usa. La compilación de los ficheros del programa y del módulo se hace con total independencia (lenguaje C ANSI con prototipos, C++, y algunas versiones del lenguaje Pascal, con la directiva EXTERN o USE).
(c) La interfaz del módulo y su realización se escriben en ficheros separados. El mismo fichero de interfaz se usa tanto al compilar la realización del módulo como al compilar el programa que lo usa (lenguajes Modula-2 y Ada).
(d) La interfaz del módulo y su realización se combinan en un solo fichero fuente. Al compilar el programa que lo usa el compilador lee el fichero fuente del módulo, pero sólo utiliza los elemento de la interfaz (lenguajes Oberon y Java).

En los lenguajes que usan la técnica (a) no hay compilación segura. En los mencionados en (b) la seguridad es mayor, pero aún hay posibilidad de errores si no coincide la interfaz del módulo con la copia usada en el programa. Con los lenguajes que usan las técnicas (c) y (d) la compilación es completamente segura.

El lenguaje **C±** está basado en C++ y comparte sus características en cuanto a compilación separada y compilación segura.

### 15.1.3 Descomposición modular

La posibilidad de compilar módulos de forma separada permite repartir el trabajo de desarrollo de un programa, a base de realizar su *descomposición modular*. Los diferentes módulos pueden ser encargados a programadores diferentes, y gracias a ello todos pueden trabajar al mismo tiempo.

De esta forma se pueden desarrollar en un tiempo razonable los grandes programas correspondientes a las aplicaciones de hoy día, que totalizan cientos de miles o millones de sentencias.

La descomposición modular de un programa puede reflejarse en un diagrama de estructura, tal como el de la figura 15.2. En este diagrama se representa cada módulo como un rectángulo, con el nombre del módulo en su interior, y se usan líneas para indicar las relaciones de uso entre ellos.

En este ejemplo el módulo A usa elementos de los módulos B y C, y el módulo B usa elementos de C y D. Los módulos C y D no usan ningún otro módulo.

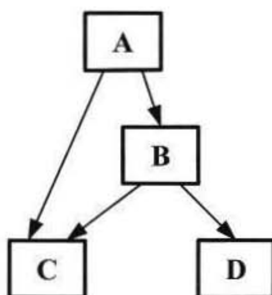



Figura 15.2 Ejemplo de diagrama de estructura.

Las líneas que indican relaciones de uso pueden llevar punta de flecha si es necesario indicar expresamente cuál es el sentido de la relación. Normalmente no es necesario, pues, como en este caso, un módulo que usa otro se dibuja encima de él, de manera que las líneas de uso se interpretan siempre de arriba a abajo, estableciendo al mismo tiempo una jerarquía entre módulos.

El objetivo de la ingeniería de software es facilitar el desarrollo de una aplicación de forma organizada, de manera que muchas personas puedan colaborar simultáneamente en un mismo proyecto. Para que la descomposición en módulos sea adecuada, desde ese punto de vista, conviene que los módulos resulten tan independientes unos de otros como sea posible. Esta independencia se analiza según dos criterios, denominados *acoplamiento* y *cohesión*.

El *acoplamiento* entre módulos indica cuántos elementos distintos o características de uno o varios módulos han de ser tenidos en cuenta a la vez al usar un módulo desde otro. Este acoplamiento debe reducirse a un mínimo.

La *cohesión* indica el grado de relación que existe entre los distintos elementos de un mismo módulo, y debe ser lo mayor posible. Esto quiere decir que dos elementos íntimamente relacionados deberían ser definidos en el mismo módulo, y que un mismo módulo no debe incluir elementos sin relación entre sí.

## 15.2 Módulos en C±

Un programa descompuesto en módulos se escribe como un conjunto de *ficheros fuente* relacionados entre sí, y que pueden compilarse por separado. Cada fichero fuente constituye así una *unidad de compilación*.

Lamentablemente hay que decir que los lenguajes C o C++ (y por tanto **C±**) no incorporan ninguna estructura sintáctica para realizar programación modular. Las descomposición de un programa en partes se hace solamente a nivel físico combinando ficheros, y no a nivel lógico, usando estructuras bien definidas. Por esa razón se debe imponer una cierta disciplina de codificación que permita mantener la organización modular del programa dentro de unos límites aceptables de esfuerzo de comprensión y mantenibilidad.

Las siguientes secciones indican la manera de redactar programas compuestos por varios módulos, señalando la forma de hacer corresponder los módulos lógicos con ficheros físicos de código fuente, y la manera de compilar el programa en su conjunto.

### 15.2.1 Proceso de compilación simple

Un *fichero fuente* es un fichero de texto que contiene el código de una *unidad de compilación*, es decir, es posible invocar el compilador dándole como entrada sólo ese fichero fuente.

La compilación de un fichero fuente produce un fichero objeto que contiene la traducción del código **C±** a instrucciones de máquina, tal como se representa en el ejemplo de la figura 15.3. Por convenio, los ficheros fuente en **C±** tienen la extensión `.cpp` (la misma usada habitualmente en C++) y los ficheros objeto la extensión `.o`. Como regla de disciplina modular en **C±** se exige que el nombre del fichero objeto sea el mismo que el del fichero fuente.

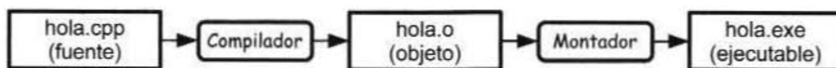



Figura 15.3 Proceso de compilación simple.

■ NOTA: El lenguaje **C±** es un subconjunto estricto del lenguaje C++. No existe un compilador específico para **C±**. Los ficheros fuente en **C±** se compilarán con un compilador de C++.

En general un fichero objeto no se puede ejecutar directamente. Se necesita un paso adicional de montaje para obtener un programa o fichero ejecutable. En MS-Windows los ficheros ejecutables tienen la extensión `.exe` (en UNIX/Linux no suelen tener extensión). Si el programa ejecutable se ha generado a partir de un solo fichero fuente, debe tener también el mismo nombre.

En C y C++ es frecuente que el montador y el compilador sean una misma herramienta, o al menos que se invoquen como si lo fueran. En casos sencillos como éste es posible realizar la compilación y montaje como una sola operación.

En esta primera visión simplificada del proceso de compilación y montaje se ha omitido mencionar explícitamente las librerías estándar o de sistema. Más adelante se mostrará cómo se usan durante ese proceso.

### 15.2.2 Módulo principal

Cuando se descompone un programa en **C±** en varios módulos uno de ellos ha de ser el *programa principal* o *módulo principal*. Este módulo será el que contenga la función `main()`. La ejecución del programa completo equivale a la ejecución de dicha función principal. Por supuesto, la función principal puede invocar durante su ejecución operaciones definidas en otros módulos. Repetiremos aquí el primer ejemplo de programa completo mostrado en este libro, y que corresponde también al ejemplo de la figura 15.3:

```
/** Programa: Hola */
/* Este programa escribe Hola */

#include <stdio.h>

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

Todos los ejemplos de programas completos desarrollados hasta ahora se componían exclusivamente de un módulo principal. Para ser precisos habrá que decir que en estos ejemplos se usaban otros módulos de las librerías estándar, tal como `stdio`, pero puede considerarse que estos módulos de sistema no son parte del programa o aplicación desarrollada.

La manera de escribir un módulo principal ya ha sido expuesta en los temas anteriores. Sólo falta indicar que el fichero fuente de ese programa principal debe tener el nombre que se dará finalmente al programa ejecutable y tener la extensión `.cpp`, tal como se ha indicado anteriormente.

### 15.2.3 Módulos no principales

Los módulos de la aplicación que no contienen una función `main()` no permiten generar un programa ejecutable por sí solos. Los elementos que contienen están destinados a ser usados por el programa principal u otros módulos. Al escribir el código de estos módulos no principales hay que distinguir claramente entre los *elementos públicos*, que deben ser visibles desde fuera del módulo para

poder usarlos, y los *elementos privados*, que sólo necesitan ser visibles en el interior del módulo.

La distinción entre los elementos públicos y los privados se hace repartiendo el código del módulo en dos ficheros fuente separados: un *fichero de interfaz* o *fichero de cabecera*, y un *fichero de implementación*. El siguiente ejemplo muestra el código de un módulo que ofrece facilidades para imprimir series de valores numéricos tabulando en varias columnas. El fichero de interfaz es:

```
/************************************************
* Interfaz de módulo: Tabulacion
*
*   Este módulo contiene los elementos para
*   imprimir series de números en varias columnas
************************************************/
#pragma once

extern int numColumnas;   /* número de columnas */
extern int anchoColumna;  /* ancho de cada una */

/*-- Iniciar la impresión --*/
void Iniciar( char titulo[] );

/* Imprime un número, tabulando */
void Imprimir( int numero );

/* Completa la impresión de la última línea */
void Terminar();
```

A continuación se presenta el correspondiente fichero de implementación. En este ejemplo de código se ha aprovechado una característica especial del procedimiento `printf` que permite indicar el ancho de un campo como argumento además del valor a imprimir, usando un asterisco como especificador del ancho en el formato.

```
/************************************************
* Módulo: Tabulacion
*
*   Este módulo contiene los elementos para
*   imprimir series de números en varias columnas
************************************************/
#include <stdio.h>
#include <string.h>
#include "Tabulacion.h"
```

```
int numColumnas = 4;      /* número de columnas */
int anchoColumna = 10;   /* ancho de cada una */

static int columna = 1;   /* columna actual */

/*-- Iniciar la impresión --*/
void Iniciar( char titulo[] ) {
  Terminar(); /* la serie anterior, por si acaso */
  printf( "%s\n", titulo );
  columna = 1;
}

/*-- Imprime un número, tabulando --*/
void Imprimir( int numero ) {
  if (columna > numColumnas) {
    printf( "\n" );
    columna = 1;
  }
  printf( "%*d", anchoColumna, numero );
  columna++;
}

/*-- Completar la impresión de la última línea --*/
void Terminar() {
  if ( columna > 1 ) {
    printf( "\n" );
  }
  columna = 1;
}
```

El código contiene nuevos elementos de **C±** (`#pragma once`, `extern`, `static`) que se explican más adelante.

Una buena disciplina de nombres exige que ambos ficheros tengan el mismo nombre que el nombre lógico del módulo. El fichero de interfaz tendrá la extensión `.h` y el de implementación la extensión `.cpp`. Por lo tanto los nombres de los ficheros fuente en este ejemplo deben ser:

- Interfaz: `Tabulacion.h`
- Implementación: `Tabulacion.cpp`

En **C±** no existe el concepto de nombre de módulo a nivel de sintaxis del lenguaje. El nombre lógico `Tabulacion` usado en el ejemplo aparece solamente en comentarios de documentación del código. La directiva `#include` sirve para hacer referencia a un fichero fuente desde otro, y tiene como parámetro el nombre del fichero físico `"Tabulacion.h"`, incluyendo la extensión.

### 15.2.4 Uso de módulos

Para usar los elementos públicos definidos en un módulo hay que incluir la interfaz de ese módulo en el código donde se vaya a utilizar. Esto se consigue con la directiva `#include` que ya se ha empleado en otros ejemplos para usar las librerías estándar. La novedad ahora es que los nombres de los ficheros de la propia aplicación deben escribirse entre comillas (`"..."`) y no entre ángulos (`<...>`). Con esto se indica al compilador que debe buscar dichos ficheros en donde reside el código fuente de la aplicación y no donde está instalada la herramienta de compilación. Ejemplo:

```
/***********************************************
* Programa: Serie
*
*   Este programa imprime la serie de números
*   del 1 al 20 en varias columnas
***********************************************/
#include "Tabulacion.h"

int main() {

  Iniciar( "-- Columnas por defecto --" );
  for (int k = 1; k <= 20; k++) {
    Imprimir(k);
  }
  Terminar();

  numColumnas = 3;
  anchoColumna = 13;
  Iniciar( "-- 3 columnas de 13 caracteres --" );
  for (int k = 1; k <= 20; k++) {
    Imprimir(k);
  }
  Terminar();

  numColumnas = 6;
  anchoColumna = 5;
  Iniciar( "-- 6 columnas de 5 caracteres --" );
  for (int k = 1; k <= 20; k++) {
    Imprimir(k);
  }
  Terminar();
}
```

Este programa imprime una serie de números correlativos, usando el módulo de tabulación anterior, y con varias configuraciones de columnas. El resultado es:

```
-- Columnas por defecto --
         1         2         3         4
         5         6         7         8
         9        10        11        12
        13        14        15        16
        17        18        19        20
-- 3 columnas de 13 caracteres --
            1            2            3
            4            5            6
            7            8            9
           10           11           12
           13           14           15
           16           17           18
           19           20
-- 6 columnas de 5 caracteres --
    1    2    3    4    5    6
    7    8    9   10   11   12
   13   14   15   16   17   18
   19   20
```

### 15.2.5 Declaración y definición de elementos públicos

En los ejemplos de programas de los temas anteriores los distintos elementos creados por el programador se definían completamente en un punto determinado del código. En los programas modulares, en los que hay elementos de un módulo que se usan en otros, es preciso distinguir a veces entre la *declaración* y la *definición* de un elemento.

En la *declaración* de un elemento hay que especificar lo necesario para que el compilador pueda compilar correctamente el código que usa dicho elemento.

En la *definición* de un elemento hay que especificar lo necesario para que el compilador genere el código del propio elemento.

En el caso de los elementos públicos de los módulos, la declaración debe ponerse en el fichero de interfaz, y la definición en el fichero de implementación. En algunos casos la declaración contiene toda la información posible, y no hace falta una definición complementaria. La siguiente tabla recoge un resumen de cómo se declaran y definen en **C±** las distintas clases de elementos sintácticos.

| **Declaración (`fichero.h`)** | **Definición (`fichero.cpp`)** |
|---|---|
| `typedef ...` *TipoNuevo* `...;` | (no aplicable) |
| `const` *Tipo constante* `=` *valor*`;` | (no aplicable) |
| `extern` *Tipo variable*`;` | *Tipo variable* `=` *valor*`;` |
| *Tipo Subprograma*`(`*argumentos*`);` | *Tipo Subprograma*`(`*argumentos*`) {`<br>`...` *código* `...`<br>`}` |

- Los tipos y constantes se especifican totalmente en el fichero de interfaz. No hay declaración y definición separadas.
- Las variables se definen de la manera habitual en el fichero de implementación, incluyendo la especificación de valor inicial en su caso. Con ello el compilador reserva espacio para dicha variable en el módulo que la define. En el fichero de interfaz se pone además una declaración que indica el tipo y nombre de la variable, sin indicar valor inicial, y precedida de la palabra clave `extern`. Esta declaración permite al compilador generar código de las sentencias que usan dicha variable en otros módulos sin reservar espacio para ella, ya que formará parte efectiva del código del módulo que la define. La conexión entre las referencias a la variable y su ubicación real se resuelve durante la fase de montaje, posterior a la compilación.
- Los subprogramas se definen de la manera habitual en el fichero de implementación y permiten al compilador generar el código objeto del subprograma. En el fichero de interfaz se pone además una declaración en forma de *prototipo* o *cabecera de subprograma* sólo con el tipo, nombre y argumentos. Esta cabecera permite al compilador generar el código de las sentencias de llamada al subprograma. La conexión entre las llamadas al subprograma y su código real se resuelve durante la fase de montaje, posterior a la compilación.

### 15.2.6 Conflicto de nombres en el ámbito global

El ámbito más externo en la jerarquía de bloques del programa principal y de todos los módulos de una aplicación constituye un espacio de nombres global y único, en el que no debe haber nombres repetidos. Esto exige una clara disciplina para evitar conflictos debidos al intento de usar el mismo identificador en módulos diferentes al definir elementos distintos. Por ejemplo, si dos módulos diferentes definen cada uno una operación de inicialización, y ambos le dan el nombre `Iniciar()`, se obtendrá un error al tratar de compilar y/o montar un programa que use ambos módulos.

■ NOTA: En C++ es posible a veces definir subprogramas diferentes con el mismo nombre, si se distinguen claramente por su tipo y número de argumentos. Esta facilidad se denomina "*sobrecarga*" (*overload*). En **C±** no se admite dicha posibilidad.

Una técnica sencilla para evitar en lo posible los conflictos de nombres públicos globales es asignar a cada módulo un prefijo diferente que se habrá de usar en los nombres de todos sus elementos públicos. De esta manera no hay que atender a cada nombre particular para evitar conflictos, sino que basta controlar que no haya prefijos repetidos. En el ejemplo anterior del módulo de tabulación se podría haber empleado el prefijo `TAB` en los nombres de las operaciones públicas:

```
/************************************************
* Interfaz de módulo: Tabulacion (TAB)
************************************************/
#pragma once

extern int TAB_numColumnas;   /* número de columnas */
extern int TAB_anchoColumna;  /* ancho de cada una */

/*-- Iniciar la impresión --*/
void TAB_iniciar( char titulo[] );

/* Imprime un número, tabulando */
void TAB_imprimir( int numero );

/* Completa la impresión de la última línea */
void TAB_terminar();
```

■ NOTA: El estándar actual del lenguaje C introduce el mecanismo de "*espacios de nombres*" (*namespaces*) que tiene posibilidades similares al empleo de prefijos, pero de forma más organizada (y también algo más complicada de usar).

El ámbito global más externo incluye también los elementos privados, que no figuran en la interfaz de los módulos. Afortunadamente en este caso el lenguaje **C±** ofrece un mecanismo que evita los conflictos de nombres repetidos, ya que es posible especificar elementos en el ámbito más externo que sólo sean visibles en el fichero fuente donde se definen. Para ello basta poner la palabra clave `static` delante de la definición del elemento. Esto es lo que se ha hecho en el ejemplo de tabulación con la variable auxiliar que almacena el estado del proceso de impresión en varias columnas:

```
/************************************************
* Módulo: Tabulacion
************************************************/

static int columna = 1;   /* columna actual */
```

Con esta definición es posible reutilizar el nombre `columna` para elementos globales de otros módulos, sin que haya conflicto entre ellos.

### 15.2.7 Unidades de compilación en C±

Los ficheros fuente de los tipos mencionados pueden ser considerados unidades de compilación, en el sentido de que es posible invocar la compilación de cada uno de ellos por separado. Por lo tanto tendremos como unidades de compilación:

- El módulo principal: *programa*.`cpp`
- El fichero de interfaz de un módulo: *modulo*.`h`
- El fichero de implementación de un módulo: *modulo*.`cpp`

En realidad a la hora de preparar una aplicación sólo se mandan compilar realmente los ficheros con extensión `.cpp`, que son los que generan código objeto. Los fichero de interfaz con extensión `.h` no se mandan compilar por sí mismos, ya que en principio no generan código objeto. De hecho algunos compiladores de C y C++ rechazan la compilación de este tipo de ficheros, incluso aunque el código que contienen sea perfectamente aceptable y compilable como fichero con extensión `.cpp`.

Lo que ocurre es que los ficheros de interfaz son parte efectiva de la compilación de los ficheros de implementación. El significado de la directiva `#include` es equivalente a copiar en ese punto el contenido del fichero fuente indicado. Esta copia o inclusión se hace sobre la marcha durante la compilación, en una fase inicial de la misma denominada *preproceso*. La figura 15.4 muestra cómo se preprocesa un módulo o programa de nombre A que usa otro de nombre B.

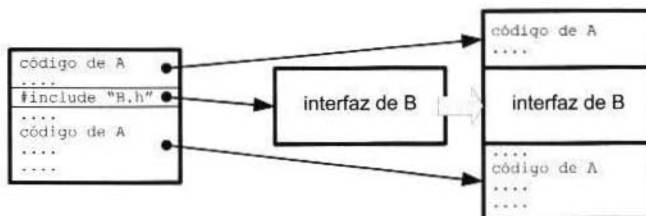



Figura 15.4 Expansión de la directiva `#include` durante el preproceso.

Finalmente se enumeran aquí las reglas de sintaxis correspondientes a la estructura general de cada una de las unidades de compilación mencionadas:

*Unidad_de_compilación* **::=**
*Programa_principal* **|** *Módulo_interfaz* **|** *Módulo_implementación*

```
Programa_principal ::= { Include } { Declaración_global }
      int main() Bloque
Módulo_interfaz ::= Pragma_once { Include } { Declaración_interfaz }
Módulo_implementación ::= { Include } { Declaración_global }
Include ::=
      #include <Nombre_módulo.h> | #include "Nombre_módulo.h"
Pragma_once ::= #pragma once
Declaración_global ::=
      Declaración_de_constante |
      Declaración_de_tipo |
      [ static ] Declaración_de_variable |
      [ static ] Subprograma
Declaración_interfaz ::=
      Declaración_de_constante |
      Declaración_de_tipo |
      Declaración_de_variable_externa |
      Cabecera_subprograma ;
Declaración_de_variable_externa ::=
      extern Identificador_de_tipo Lista_de_identificadores ;
```

### 15.2.8 Compilación de programas modulares. Proyectos

El proceso de compilación y montaje de un programa cuyo código fuente está repartido entre varios módulos requiere una cadena de operaciones, tal como se indica en la figura 15.5 para el ejemplo del programa de tabulación.

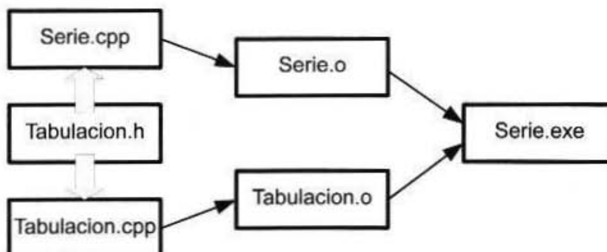



Figura 15.5 Compilación y montaje de un programa modular.

Es importante observar que el fichero de interfaz `Tabulacion.h` se incluye tanto en el programa principal `Serie.cpp` como en el propio fichero de implementación `Tabulación.cpp`. Eso es necesario para asegurar una *compilación*

*segura* al poder detectarse posibles errores de codificación que hagan inconsistente la definición de los elementos de un módulo con el uso que se hace de ellos desde otras partes del programa.

La generación del programa ejecutable final exige:

1. Compilar los módulos uno a uno, generando el correspondiente fichero objeto (`.o`) a partir del fuente (`.cpp`). Cada compilación individual usa también los ficheros de interfaz (`.h`) mencionados en las directivas `#include` del módulo.
2. Montar el programa ejecutable combinando todos los ficheros objeto de los módulos.

Los entornos de programación modernos simplifican la tarea de recompilar los módulos después de editar alguno o varios de ellos. Para eso disponen de un mecanismo de *proyectos*, consistente en disponer de un fichero con información de los ficheros fuente que forman parte de la aplicación. A partir de ahí el entorno automatiza la generación o actualización de los ficheros objeto y el ejecutable final, que se invoca en conjunto como una operación única denominada habitualmente "*construir*" (en inglés *build*).

La información mínima que debe contener el fichero de descripción de un proyecto será:

- Nombre del proyecto (= nombre del programa ejecutable)
- Lista de ficheros fuente de implementación `.cpp` (incluyendo el programa principal)

Opcionalmente se puede disponer también de otros datos útiles, tales como

- Lista de ficheros de interfaz `.h`
- Forma de invocar al compilador, con opciones particulares para ese proyecto
- etc.

## 15.3 Desarrollo modular basado en abstracciones

La organización de un programa en módulos puede hacerse aplicando diferentes criterios, en función de cada caso concreto. En general será adecuado cualquier criterio que conduzca a módulos con buena cohesión y bajo acoplamiento. El reconocimiento de abstracciones en el desarrollo de una aplicación es al mismo tiempo una ayuda para elegir una organización modular adecuada.

### 15.3.1 Implementación de abstracciones como módulos

En la mayoría de los casos los tipos *abstractos de datos* identificados en una aplicación son buenos candidatos para ser codificados como módulos independientes, y lo mismo ocurre con las *abstracciones funcionales* de cierta complejidad. Por lo tanto el desarrollo basado en abstracciones lleva implícita una posible descomposición natural del programa en módulos. Esto ocurre de forma obligada en algunos lenguajes con una estricta orientación a objetos, por ejemplo Java.

Para ilustrar esta idea se repite aquí el ejemplo de dibujar una Curva-C introducido en el tema 14, pero dedicando ahora un módulo separado a cada abstracción principal.

```
/****************************************************************
* Interfaz de módulo: Papel
*
*   Este módulo define el tipo abstracto PAPEL, capaz de
*   almacenar un dibujo formado por trazos en una cuadrícula
****************************************************************/
#pragma once

const int ANCHO = 32;  /* cuadrícula en */
const int ALTO = 19;   /* la pantalla */

typedef struct TipoPapel {
  void PonerEnBlanco();
  void MarcarHorizontal( int x, int y );
  void MarcarVertical( int x, int y );
  void Imprimir();
private:
  bool Dentro( int x, int y );
  char marcasH[ANCHO][ALTO];
  char marcasV[ANCHO][ALTO];
};
```

```
/****************************************************************
* Módulo: Papel
*
*   Este módulo define el tipo abstracto PAPEL, capaz de
*   almacenar un dibujo formado por trazos en una cuadrícula
****************************************************************/
#include <stdio.h>
#include "Papel.h"
```

```
bool TipoPapel::Dentro( int x, int y ) {
  return (x >= 0 && x < ANCHO && y >= 0 && y < ALTO);
}

void TipoPapel::PonerEnBlanco() {
  for (int x=0; x<ANCHO; x++) {
    for (int y=0; y<ALTO; y++) {
      marcasH[x][y] = ' ';
      marcasV[x][y] = ' ';
    }
  }
}

void TipoPapel::MarcarHorizontal( int x, int y ) {
  if (Dentro( x, y )) {
    marcasH[x][y] = '_';
  }
}

void TipoPapel::MarcarVertical(int x, int y ) {
  if (Dentro( x, y )) {
    marcasV[x][y] = '|';
  }
}

void TipoPapel::Imprimir() {
  for (int y=ALTO-1; y>=0; y--) {
    for (int x=0; x<ANCHO; x++) {
      printf( "%c%c", marcasV[x][y], marcasH[x][y]);
    }
    printf( "\n" );
  }
}
```

```
/*******************************************************************
* Interfaz de módulo: Tortuga
*
*   Este módulo define el tipo abstracto TORTUGA, capaz de
*   ir trazando una trayectoria mediante avances y giros
*******************************************************************/
#pragma once
#include "Papel.h"

typedef enum TipoRumbo { Este, Norte, Oeste, Sur };
```

```
typedef struct TipoTortuga {
  void Poner( int x, int y, TipoRumbo rumbo );
  void Avanzar( TipoPapel & p );
  void GirarDerecha();
  void GirarIzquierda();
private:
  int xx, yy;
  TipoRumbo sentido;
};
```

```
/****************************************************************
* Módulo: Tortuga
*
*   Este módulo define el tipo abstracto TORTUGA, capaz de
*   ir trazando una trayectoria mediante avances y giros
****************************************************************/
#include "Tortuga.h"

void TipoTortuga::Poner( int x, int y, TipoRumbo rumbo ) {
  xx = x;
  yy = y;
  sentido = rumbo;
}

void TipoTortuga::Avanzar( TipoPapel & p ) {
  switch (sentido) {
  case Norte:
    p.MarcarVertical( xx, yy );
    yy++;
    break;
  case Sur:
    yy--;
    p.MarcarVertical( xx, yy );
    break;
  case Este:
    p.MarcarHorizontal( xx, yy );
    xx++;
    break;
  case Oeste:
    xx--;
    p.MarcarHorizontal( xx, yy );
    break;
  }
}
```

```
void TipoTortuga::GirarDerecha() {
  sentido = TipoRumbo( (int(sentido)-1+4) % 4 );
                        /* +4 evita rumbo negativo */
}

void TipoTortuga::GirarIzquierda() {
  sentido = TipoRumbo( (int(sentido)+1) % 4 );
}
```

```
/*****************************************************************
* Interfaz de módulo: CurvaC
*
*   Este módulo contiene la función que genera una Curva-C
*****************************************************************/
#pragma once
#include "Tortuga.h"
#include "Papel.h"

void CurvaC( int orden, TipoTortuga & t, TipoPapel & p );
```

```
/*****************************************************************
* Módulo: CurvaC
*
*   Este módulo contiene la función que genera una Curva-C
*****************************************************************/
#include "CurvaC.h"

void CurvaC( int orden, TipoTortuga & t, TipoPapel & p ) {
  if (orden == 0) {
    t.Avanzar( p );
  } else if (orden > 0) {
    CurvaC( orden - 1, t, p );
    t.GirarDerecha();
    CurvaC( orden - 1, t, p );
    t.GirarIzquierda();
  }
}
```

```
/*****************************************************************
* Programa: DibujarC
*
* Descripción:
*   Este programa lee como dato el orden de una curva C, y a
*   continuación la dibuja en forma de texto en pantalla
*****************************************************************/
```

```
#include <stdio.h>
#include "CurvaC.h"
#include "Tortuga.h"
#include "Papel.h"

int main() {
  int orden;
  TipoTortuga tt;
  TipoPapel pp;

  printf( "Orden: " );
  scanf( "%d", &orden );

  pp.PonerEnBlanco();
  /* posición inicial de conveniencia */
  tt.Poner( 8, 3, Este );
  CurvaC( orden, tt, pp );
  pp.Imprimir();
}
```

Como puede verse, los ficheros de interfaz contienen obviamente la declaración de la interfaz del elemento abstracto, y los ficheros de implementación contienen la realización del elemento. En el caso de la abstracción funcional CurvaC la interfaz es simplemente la cabecera del subprograma.

En el caso de los tipos abstractos de datos Tortuga y Papel, la interfaz incluye no sólo el tipo `struct` que lo define, sino también los complementos necesarios. En la interfaz de la abstracción Tortuga se ha incluido la definición del tipo enumerado `TipoRumbo`, y en la del Papel se incluyen las constantes de tamaño.

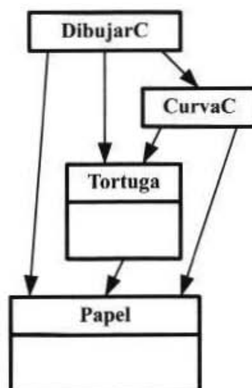



Figura 15.6 Estructura modular del programa de dibujar una Curva-C.

La figura 15.6 muestra la estructura del programa, indicando las relaciones de dependencia (de uso) de unos módulos respecto a otros. Las flechas indican que un módulo utiliza directamente elementos de otro.

En el diagrama, cada módulo se representa, en general, como un rectángulo. Además, los módulos que corresponden a tipos abstractos de datos se han representado de manera similar a la usada en los diagramas de clases UML, marcando una banda superior con el nombre del módulo. Se puede aumentar la analogía con la *notación UML* enumerando en la banda inferior los elementos públicos del módulo, como se muestra en la figura 15.7:

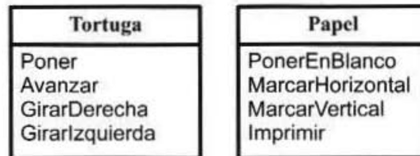



Figura 15.7 Representación gráfica de la interfaz de los módulos.

## 15.3.2 Dependencias entre ficheros. Directivas

Las relaciones de uso entre módulos se corresponden, en principio, con las directivas `#include` usadas en un fichero fuente para hacer visibles los elementos de otro, y que pueden aparecer en el fichero `.cpp` y/o en el `.h`. La recomendación es:

- Un fichero `xxx.h` debe incluir otros `yyy.h` que use directamente.
- Un fichero `xxx.cpp` debe incluir su propio `xxx.h` y otros `yyy.h` que use directamente. Pero no hace falta hacerlo explícitamente si ya los incluye su `xxx.h`.

En el ejemplo anterior hay dependencias indirectas. Por ejemplo, el módulo principal `DibujarC` usa elementos de `Tortuga` directamente y también indirectamente a través de `CurvaC`. Si no se toman precauciones el preprocesador incluirá el código de `Tortuga.h` dos veces, y se tendrán errores por duplicación de definiciones. La directiva `#pragma once` sirve precisamente para evitar esa duplicación.

■ NOTA: La directiva `#pragma once` no es estándar en C++. En su lugar la receta habitual es similar a la siguiente (para un fichero `xxx.h`):

```
#ifndef xxx_h_
#define xxx_h_
... código de la interfaz ...
#endif
```

Este esquema de código es algo más seguro pero bastante más complejo, y por ello se ha decidido no incluirlo en el subconjunto **C±**.

### 15.3.3 Datos encapsulados

En este tema y el anterior se ha visto cómo al definir un tipo abstracto de datos hay que declarar luego variables de ese tipo para trabajar con ellas. En algunos casos concretos resulta que sólo es necesario una única variable del tipo abstracto. Si es así, existe la posibilidad de encapsular dicha variable en un módulo y evitar la declaración explícita del tipo. La facilidad de ocultación que provee la programación modular es suficiente para conseguir la *abstracción del dato*, de forma que sólo sean visibles las operaciones que lo manipulan pero no los detalles de su implementación.

La siguiente tabla compara los esquemas generales de código correspondientes a la declaración y uso de un tipo abstracto de datos y a un dato encapsulado. En ambos casos se usa un módulo separado para el elemento abstracto.

| Tipo abstracto | Dato encapsulado |
|---|---|
| **Interfaz** | |
| `typedef struct Tipo {`<br>`  void Operacion1();`<br>`  void Operación2();`<br>`private:`<br>`  UnTipo valorInterno;`<br>`  void Operación3();`<br>`};` | `void Operacion1();`<br>`void Operacion2();` |
| **Implementación** | |
| `void Tipo::Operacion3() {`<br>`   ...`<br>`}`<br><br>`void Tipo::Operacion1() {`<br>`   ... valorInterno ...`<br>`}`<br>`void Tipo::Operacion2() {`<br>`  ... valorInterno ...`<br>`}` | `static UnTipo valorInterno;`<br>`static void Operacion3() {`<br>`   ...`<br>`}`<br><br>`void Operacion1() {`<br>`   ... valorInterno ...`<br>`}`<br>`void Operacion2() {`<br>`   ... valorInterno ...`<br>`}` |
| **Uso** | |
| `Tipo dato;`<br><br>`dato.Operacion1();` | `Operacion1();` |

Conviene recordar que los nombres de variables y subprogramas definidos en el nivel más externo de un fichero fuente son globales, por defecto. Para que sean tratados como identificadores locales al fichero deben ser marcados como `static`.

Como puede verse el segundo esquema es más sencillo, ya que ni el tipo ni el dato son visibles. Eso es posible por la limitación de que sólo hay un dato del tipo abstracto. El dato encapsulado aparece simplemente como una colección de operaciones que manipulan la misma variable interna, oculta.

Podemos ilustrar esta técnica recodificando el ejemplo modular de dibujar una Curva-C, convirtiendo ahora el **Papel** en un dato encapsulado. Puesto que cambia el código de la interfaz, hay que retocar también el código de los demás módulos que lo usan.

```
/*****************************************************************
* Interfaz de módulo: Papel2
*
*   Este módulo encapsula un único ejemplar de dato abstracto
*   de tipo PAPEL, capaz de almacenar un dibujo formado
*   por trazos en una cuadrícula
*****************************************************************/
#pragma once

void PonerEnBlanco();
void MarcarHorizontal( int x, int y );
void MarcarVertical( int x, int y );
void Imprimir();
```

```
/*****************************************************************
* Módulo: Papel2
*
*   Este módulo encapsula un único ejemplar de dato abstracto
*   de tipo PAPEL, capaz de almacenar un dibujo formado
*   por trazos en una cuadrícula
*****************************************************************/
#include <stdio.h>
#include "Papel2.h"

/*---------- Elementos privados ----------*/

const int ANCHO = 32;  /* cuadrícula en */
const int ALTO = 19;   /* la pantalla */

typedef char MatrizMarcas[ANCHO][ALTO];
```

```
static MatrizMarcas marcasH;
static MatrizMarcas marcasV;

static bool Dentro( int x, int y ) {
  return (x >= 0 && x < ANCHO && y >= 0 && y < ALTO);
}

/*---------- Elementos públicos ----------*/

void PonerEnBlanco() {
  for (int x=0; x<ANCHO; x++) {
    for (int y=0; y<ALTO; y++) {
      marcasH[x][y] = ' ';
      marcasV[x][y] = ' ';
    }
  }
}

void MarcarHorizontal( int x, int y ) {
  if (Dentro( x, y )) {
    marcasH[x][y] = '_';
  }
}

void MarcarVertical(int x, int y ) {
  if (Dentro( x, y )) {
    marcasV[x][y] = '|';
  }
}

void Imprimir() {
  for (int y=ALTO-1; y>=0; y--) {
    for (int x=0; x<ANCHO; x++) {
      printf( "%c%c", marcasV[x][y], marcasH[x][y]);
    }
    printf( "\n" );
  }
}
```

```
/****************************************************************
* Interfaz de módulo: Tortuga2
*
*   Este módulo define el tipo abstracto TORTUGA, capaz de
*   ir trazando una trayectoria mediante avances y giros
****************************************************************/
```

```
#pragma once

typedef enum TipoRumbo { Este, Norte, Oeste, Sur };

typedef struct TipoTortuga {
  void Poner( int x, int y, TipoRumbo rumbo );
  void Avanzar();
  void GirarDerecha();
  void GirarIzquierda();
private:
  int xx, yy;
  TipoRumbo sentido;
};
```

```
/*****************************************************************
* Módulo: Tortuga2
*
*   Este módulo define el tipo abstracto TORTUGA, capaz de
*   ir trazando una trayectoria mediante avances y giros
*****************************************************************/
#include "Tortuga2.h"
#include "Papel2.h"

void TipoTortuga::Poner( int x, int y, TipoRumbo rumbo ) {
  xx = x;
  yy = y;
  sentido = rumbo;
}

void TipoTortuga::Avanzar() {
  switch (sentido) {
  case Norte:
    MarcarVertical( xx, yy );
    yy++;
    break;
  case Sur:
    yy--;
    MarcarVertical( xx, yy );
    break;
  case Este:
    MarcarHorizontal( xx, yy );
    xx++;
    break;
  case Oeste:
    xx--;
```

```
    MarcarHorizontal( xx, yy );
    break;
  }
}

void TipoTortuga::GirarDerecha() {
  sentido = TipoRumbo( (int(sentido)-1+4) % 4 );
  /* +4 evita rumbo negativo */
}

void TipoTortuga::GirarIzquierda() {
  sentido = TipoRumbo( (int(sentido)+1) % 4 );
}
```

```
/****************************************************************
* Interfaz de módulo: CurvaC2
*
*   Este módulo contiene la función que genera una Curva-C
****************************************************************/
#pragma once
#include "Tortuga2.h"

void CurvaC( int orden, TipoTortuga & t );
```

```
/****************************************************************
* Módulo: CurvaC2
*
*   Este módulo contiene la función que genera una Curva-C
****************************************************************/
#include "CurvaC2.h"

void CurvaC( int orden, TipoTortuga & t ) {
  if (orden == 0) {
    t.Avanzar();
  } else if (orden > 0) {
    CurvaC( orden - 1, t );
    t.GirarDerecha();
    CurvaC( orden - 1, t );
    t.GirarIzquierda();
  }
}
```

Con estos cambios, el programa principal y un ejemplo de su ejecución quedan como sigue:

```
/********************************************************************
* Programa: DibujarC2
*
* Descripción:
*   Este programa lee como dato el orden de una curva C, y a
*   continuación la dibuja en forma de texto en pantalla
********************************************************************/
#include <stdio.h>
#include "CurvaC2.h"
#include "Tortuga2.h"
#include "Papel2.h"

int main() {
  int orden;
  TipoTortuga tt;

  printf( "Orden: " );
  scanf( "%d", &orden );
  PonerEnBlanco();
  /* posición inicial de conveniencia */
  tt.Poner( 8, 3, Este );
  CurvaC( orden, tt );
  Imprimir();
}
```

```
Orden: 6





         _ _ _ _
       _|   |   |_
     _|_          |
   _| |_|        _|
  |
  |_
  |  _ _
  |_|   |
  | |_ _|
  |_
  |
  |_   _         _
    |_|_|         |
      |_         _|
        |_ _|_ _|
```

Ahora han desaparecido todas la referencias a la única variable que había del tipo **TipoPapel**, tanto en su declaración como en su uso como argumento. Además han quedado ocultas las definiciones de las constantes de tamaño del papel, que son realmente detalles de implementación, lo cual puede resultar ventajoso.

Por otra parte, las modificaciones en el código no han cambiado en modo alguno el funcionamiento global del programa, que se sigue utilizando exactamente igual que antes. La modificación del código de un programa para reorganizarlo sin cambiar su funcionalidad se denomina *refactorización*.

### 15.3.4 Reutilización de módulos

Los expertos en desarrollo de software suelen considerar que la descomposición modular basada en abstracciones es una buena metodología para desarrollar módulos con bastantes posibilidades de ser reutilizados en el futuro. Los elementos abstractos de una aplicación pueden ser de utilidad en otras aplicaciones del mismo campo de actividad, especialmente si se procura que dichos elementos abstractos correspondan a elementos reales significativos, como ocurre en los ejemplos anteriores con la tortuga y el papel, que podrán reutilizarse para componer otros dibujos con trazos rectos sobre una cuadrícula.

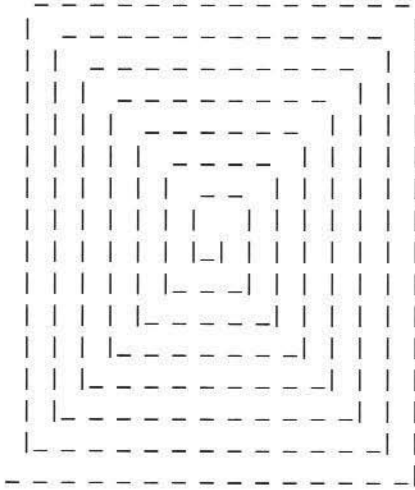
Los módulos que definen abstracciones relacionadas entre sí pueden agruparse en una biblioteca o *librería* (en inglés *library*) que se pone a disposición de quienes desarrollan aplicaciones en un campo determinado. Por ejemplo, se pueden combinar módulos que operen con fechas, horas, calendarios de diferentes culturas, políticas de cambio de horario verano/invierno, etc. Todos estos módulos constituirán una biblioteca de utilidad para quienes desarrollen aplicaciones que manejen datos de tiempo oficial.

Un caso extremo de reutilización se tiene en los módulos "*estándar*" que acompañan a muchos lenguajes de programación. Estos módulos contienen elementos de uso general, que resultan útiles en un porcentaje muy elevado de programas de todo tipo.

Como ejemplo de reutilización de **Tortuga** y **Papel** se presentan un par de programas que dibujan otras figuras. Se incluye el listado de cada uno y un ejemplo de su ejecución.

El primer ejemplo dibuja una espiral cuadrada de un tamaño dado:

```
Tamaño: 15
```

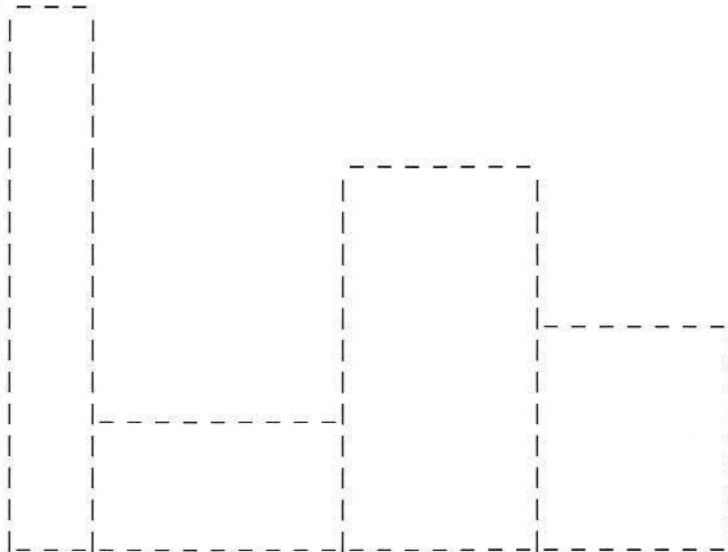


```
/*******************************************************************
* Programa: Espiral
*
* Descripción:
*   Este programa dibuja una figura espiral cuadrada, del
*   tamaño que se indique como dato
*******************************************************************/
#include <stdio.h>
#include "Tortuga.h"
#include "Papel.h"

/** Avanzar N pasos */
void AvanzarN( TipoTortuga & t, TipoPapel & p, int pasos ) {
  for (int k=1; k<=pasos; k++) {
    t.Avanzar( p );
  }
}

/** Programa principal */
int main() {
  int lado;
  TipoTortuga tt;
  TipoPapel pp;
```

```
  printf( "Tamaño: " );
  scanf( "%d", &lado );
  pp.PonerEnBlanco();
  tt.Poner( 0, 0, Este );
  for (int k=lado; k>=1; k--) {
    AvanzarN( tt, pp, k );
    tt.GirarIzquierda();
    AvanzarN( tt, pp, k );
    tt.GirarIzquierda();
  }
  pp.Imprimir();
}
```

El segundo ejemplo dibuja una serie de cajas adosadas de los tamaños que se indiquen. El final de los datos se indica mediante un tamaño cero.

```
Ancho y alto: 3 17
Ancho y alto: 9 4
Ancho y alto: 7 12
Ancho y alto: 7 7
Ancho y alto: 0 0
```



```
/*************************************************************
* Programa: Cajas
*
* Descripción:
*   Este programa dibuja una serie de cajas adosadas, de los
*   tamaños que se indiquen como datos
*************************************************************/
#include <stdio.h>
#include "Tortuga.h"
#include "Papel.h"

/** Avanzar N pasos */
void AvanzarN( TipoTortuga & t, TipoPapel & p, int pasos ) {
  for (int k=1; k<=pasos; k++) {
    t.Avanzar( p );
  }
}

/** Programa principal */
int main() {
  int ancho, alto;
  TipoTortuga tt;
  TipoPapel pp;

  pp.PonerEnBlanco();
  tt.Poner( 0, 0, Este );
  do {
    printf( "Ancho y alto: " );
    ancho = 0;
    alto = 0;
    scanf( "%d%d", &ancho, &alto );
    if (ancho > 0 && alto > 0) {
      AvanzarN( tt, pp, ancho );
      tt.GirarIzquierda();
      AvanzarN( tt, pp, alto );
      tt.GirarIzquierda();
      AvanzarN( tt, pp, ancho );
      tt.GirarIzquierda();
      AvanzarN( tt, pp, alto );
      tt.GirarIzquierda();
      AvanzarN( tt, pp, ancho );
    }
  } while (ancho > 0 && alto > 0);
  pp.Imprimir();
}
```

# Ejercicios sin resolver - III

A continuación se enuncian un tercer bloque de ejercicios sin resolver. Todos ellos deben ser realizados en **C±** utilizando la metodología explicada durante el libro. Los enunciados de los ejercicios son los siguientes:

1. Realizar un programa que simule un cajero automático de monedas. Los tipos de monedas que dispone el cajero son de 10, 20 y 50 céntimos de euro y 1 y 2 euros. Inicialmente el cajero tiene 100 monedas de cada tipo, que se van consumiendo para proporcionar las cantidades solicitadas. El cajero debe obtener la cantidad solicitada con los tipos de moneda que tenga en cada momento, tratando siempre de utilizar las monedas de mayor valor.
2. Realizar una función que a partir de dos puntos del plano pasados como argumentos, devuelva cierto cuando el primero esté mas alejado del origen de coordenadas que el segundo y falso en caso contrario. Utilizando la función anterior realizar un programa que ordene en un vector hasta 10 puntos según su distancia al origen.
3. Realizar un programa que analice un texto terminado con un punto (`.`) y extraiga del mismo las siguientes palabras:
   - Palabra más larga.
   - Palabra más corta.
   - Palabra con más vocales.
   - Palabra con más consonantes.
4. Realizar un tipo abstracto de datos (TAD) para manejar datos enteros en forma de lista con los valores ordenados de menor a mayor y que disponga de las siguientes operaciones básicas:
   - Iniciar la lista vacía.
   - Comprobar si la lista está vacía.
   - Retirar el primer número de la lista.
   - Insertar un número en la lista ordenada.
   - Conocer el número de elementos de la lista.

# Apéndice A

# Sintaxis de C±

En este apéndice se recogen de forma precisa todas las reglas sintácticas que definen el lenguaje **C±**, utilizando la *notación BNF* (Backus-Naur Form). En las reglas se utilizan los siguientes *metasímbolos*:

**::=** Metasímbolo de definición. Indica que el elemento a su izquierda puede desarrollarse según el esquema de la derecha.

**|** Metasímbolo de alternativa. Indica que puede elegirse uno y sólo uno de los elementos separados por este metasímbolo.

**{ }** Metasímbolos de repetición. Indican que los elementos incluidos dentro de ellos se pueden repetir cero o más veces.

**[ ]** Metasímbolos de opción. Indican que los elementos incluidos dentro de ellos pueden ser utilizados o no.

**( )** Metasímbolos de agrupación. Agrupan los elementos incluidos en su interior.

Estos metasímbolos se escriben con el tipo de letra especial indicado para distinguirlos de los paréntesis, corchetes, etc. que forman parte del lenguaje C+/-. También se emplearán distintos estilos de letra para distinguir los elementos simbólicos siguientes:

*Elemento_no_terminal*: Este estilo se emplea para escribir el nombre de un elemento gramatical que habrá de ser definido por alguna regla. Cualquier elemento a la izquierda del metasímbolo **::=** será no terminal y aparecerá con este estilo.

`Elemento_terminal`: Este estilo se emplea para representar los elementos que forman parte del lenguaje **C±**, es decir, que constituyen el texto de

un programa. Si aparecen en una regla deberán escribirse exactamente como se indica.

La mayoría de las reglas ya han sido introducidas a lo largo del texto. Pese a todo y con la idea de ofrecer una guía de referencia del lenguaje, se ha preferido realizar una definición conjunta de todas las reglas del lenguaje desde la regla más global hasta la más particular. Estas reglas, agrupadas por los principales elementos del lenguaje, son las siguientes:

## A.1 Unidad de compilación

1 *Unidad_de_compilación* **::=**
*Programa_principal* **|** *Módulo_interfaz* **|** *Módulo_implementación*

2 *Programa_principal* **::=** **{** *Include* **}** **{** *Declaración_global* **}**
`int main()` *Bloque*

3 *Módulo_interfaz* **::=** *Pragma_once* **{** *Include* **}** **{** *Declaración_interfaz* **}**

4 *Módulo_implementación* **::=** **{** *Include* **}** **{** *Declaración_global* **}**

## A.2 Directivas de programa

5 *Include* **::=**
`#include` <*Nombre_módulo*.h> **|** `#include` "*Nombre_módulo*.h"

6 *Pragma_once* **::=** `#pragma once`

## A.3 Declaraciones globales

7 *Declaración_global* **::=**
*Declaración_de_constante* **|**
*Declaración_de_tipo* **|**
**[** `static` **]** *Declaración_de_variable* **|**
**[** `static` **]** *Subprograma*

## A.4 Declaraciones de interfaz

8 *Declaración_interfaz* **::=**
*Declaración_de_constante* **|**
*Declaración_de_tipo* **|**
`extern` *Lista_de_variables* **|**
*Cabecera_subprograma* ;

## A.5 Constantes

9 *Declaración_de_constante* **::=** *Constante_simple*
**|** *Constante_cadena* **|** *Constante_estructurada*

10 *Constante_simple* **::=** `const` *Identificador_de_tipo*
*Identificador* = *Expresión* ;

11 *Constante_cadena* **::=** `const char`
*Identificador* [ ] = *Cadena_de_caracteres* ;

12 *Constante_estructurada* **::=** `const` *Identificador_de_tipo*
*Identificador Dimensiones* = *Inicio_estructurado* ;

13 *Dimensiones* **::=** [ *Expresión_constante* ] **{** [ *Expresión_constante* ] **}**

14 *Inicio_estructurado* **::=** { *Lista_estructurada* }

15 *Lista_estructurada* **::=** *Lista_de_valores* **|** *Lista_de_inicios*

16 *Lista_de_valores* **::=** *Expresión_constante* **{** , *Expresión_constante* **}**

17 *Lista_de_inicios* **::=** *Inicio_estructurado* **{** , *Inicio_estructurado* **}**

## A.6 Tipos

18 *Declaración_de_tipo* **::=**
*Tipo_sinónimo* **|** *Tipo_enum* **|** *Tipo_array* **|**
*Tipo_struct* **|** *Tipo_unión* **|** *Tipo_puntero*

19 *Tipo_sinónimo* **::=** `typedef` *Identificador_de_tipo Identificador* ;

20 *Tipo_enum* **::=** `typedef enum` *Identificador*
{ *Lista_de_identificadores* } ;

21 *Lista_de_identificadores* **::=** *Identificador* **{** , *Identificador* **}**

22 *Tipo_array* **::=** `typedef` *Identificador_de_tipo*
*Identificador Dimensiones* ;

23 *Tipo_struct* **::=** `typedef struct` *Identificador*
{ *Lista_de_items* **[** `private:` *Lista_de_items* **]** } ;

24 *Tipo_unión* **::=** `typedef union` *Identificador* { *Lista_de_campos* } ;

25 *Tipo_puntero* **::=** `typedef` *Identificador_de_tipo* * *Identificador* ;

26 *Lista_de_items* **::=** *Item* ; **{** *Item* ; **}**

27 *Lista_de_campos* **::=** *Campo* ; **{** *Campo* ; **}**

28 *Item* **::=** *Campo* **|** *Cabecera_subprograma*

29 *Campo* **::=** *Campos_igual_tipo* **|** *Campo_puntero* **|** *Campo_array*

30 *Campos_igual_tipo* **::=** *Identificador_de_tipo Lista_de_identificadores*

31 *Campo_puntero* **::=** *Identificador_de_tipo* * *Identificador*

32 *Campo_array* **::=** *Identificador_de_tipo Identificador Dimensiones*

## A.7 Variables

33 *Declaración_de_variable* **::=**
*Variable_simple* **|** *Variable_estructurada* **|** *Lista_de_variables*

34 *Variable_simple* **::=**
*Identificador_de_tipo Identificador* **[** = *Expresión* **]** ;

35 *Variable_estructurada* **::=**
*Identificador_de_tipo Identificador* **[** = *Inicio_estructurado* **]** ;

36 *Lista_de_variables* **::=**
*Identificador_de_tipo Lista_de_identificadores* ;

## A.8 Subprogramas

37 *Cabecera_subprograma* **::=** *Cabecera_función* **|** *Cabecera_procedimiento*

38 *Cabecera_función* **::=** *Identificador_de_tipo*
**[** *Identificador* :: **]** *Identificador* ( *Lista_de_argumentos* )

39 *Cabecera_procedimiento* **::=** `void`
**[** *Identificador* :: **]** *Identificador* ( *Lista_de_argumentos* )

40 *Lista_de_argumentos* **::=** *Argumento* **{** , *Argumento* **}**

41 *Argumento* **::=** `const` *Identificador_de_tipo Identificador* **[** [ ] **]** **|**
*Identificador_de_tipo* **[** & **]** *Identificador* **[** [ ] **]**

42 *Subprograma* **::=** *Cabecera_subprograma Bloque*

## A.9 Bloque de código

43 *Bloque* **::=** { *Secuencia_de_declaraciones Secuencia_de_sentencias* }

44 *Secuencia_de_declaraciones* **::=** **{** *Declaración_de_bloque* **}**

45 *Secuencia_de_sentencias* **::=** **{** *Sentencia* **}**

## A.10 Declaraciones de bloque

46 *Declaración_de_bloque* **::=**
*Declaración_de_constante* **|**
*Declaración_de_tipo* **|**
*Declaración_de_variable*

## A.11 Sentencias ejecutables

47 *Sentencia* **::=**
*Asignación* **|** *Incremento* **|** *Decremento* **|**
*If_else* **|** *Switch* **|**
*While* **|** *Do_while* **|**
*For_creciente* **|** *For_creciente_menor* **|** *For_decreciente* **|**
*Llamada_a_procedimiento* **|**
*Continue* **|** *Return* **|**
*Delete* **|**
*Throw* **|** *Try* **|**
*Sentencia_nula* **|**
{ *Secuencia_de_sentencias* }

48 *Asignación* **::=** *Identificador_general* = *Expresión* ;

49 *Incremento* **::=** *Identificador_general* ++ ;

50 *Decremento* **::=** *Identificador_general* -- ;

51 *If_else* **::=** `if` ( *Expresión* ) { *Secuencia_de_sentencias* }
**{** `else if` ( *Expresión* ) { *Secuencia_de_sentencias* } **}**
**[** `else` { *Secuencia_de_sentencias* } **]**

52 *Switch* **::=** `switch` ( *Expresión* ) { *Lista_de_casos* }

53 *Lista_de_casos* **::=**
*Caso* **{** *Caso* **}** **[** `default` : *Secuencia_de_sentencias* **]**

54 Caso ::= Lista_de_opciones Secuencia_de_sentencias break ;

55 Lista_de_opciones ::= case Opción : { case Opción : }

56 Opción ::= Expresión_constante

57 While ::= while ( Expresión ) { Secuencia_de_sentencias }

58 Do_while ::= do { Secuencia_de_sentencias } while ( Expresión ) ;

59 For_creciente ::= for ( int Identificador = Expresión ;
    Identificador <= Expresión ; Identificador ++ )
    { Secuencia_de_sentencias }

60 For_creciente_menor ::= for ( int Identificador = Expresión ;
    Identificador < Expresión ; Identificador ++ )
    { Secuencia_de_sentencias }

61 For_decreciente ::= for ( int Identificador = Expresión ;
    Identificador >= Expresión ; Identificador -- )
    { Secuencia_de_sentencias }

62 Llamada_a_procedimiento ::=
    Identificador_general ( Lista_de_expresiones ) ;

63 Continue ::= continue ;

64 Return ::= return [ Expresión ] ;

65 Delete ::= delete Identificador ;

66 Throw ::= throw Expresión ;

67 Try ::= try { Secuencia_de_sentencias }
    [ catch ( Identificador_de_tipo Identificador )
    { Secuencia_de_sentencias } ]

68 Sentencia_nula ::= ;

## A.12 Expresiones

69 Lista_de_expresiones ::= [ Expresión { , Expresión } ]

70 Expresión ::= Expresión_OR { Operador_OR Expresión_OR }

71 Expresión_OR ::= Expresión_AND
    { Operador_AND Expresión_AND }

72 Expresión_AND ::= Expresión_igualdad
    [ Operador_igualdad Expresión_igualdad ]

73 Expresión_igualdad ::= Expresión_numérica
    [ Operador_comparación Expresión_numérica ]

74 Expresión_numérica ::= Término { Operador_sumador Término }

75 Término ::= Factor { Operador_multiplicador Factor }

76 Factor ::= + Factor | - Factor | ! Factor | * Factor | & Factor |
    Factor_cualificado

77 Factor_cualificado ::= Elemento |
    Factor_cualificado . Elemento |
    Factor_cualificado [ Expresión ] |
    Factor_cualificado -> Elemento

78 Operador_OR ::= ||

79 Operador_AND ::= &&

80 Operador_igualdad ::= == | !=

81 Operador_comparación ::= > | >= | < | <=

82 Operador_sumador ::= + | -

83 Operador_multiplicador ::= * | / | %

## A.13 Elementos básicos

84 Elemento ::=
    Valor_entero | Valor_real | Carácter | Cadena_de_caracteres |
    Identificador | new Identificador |
    Identificador ( Lista_de_expresiones ) |
    Identificador_de_tipo ( Expresión ) |
    ( Expresión )

85 Valor_entero ::= [ + | - ] Secuencia_dígitos

86 Secuencia_dígitos ::= Dígito { Dígito }

87 Dígito ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

88 Valor_real ::= Valor_entero . [ Secuencia_dígitos ] [ Escala ]

89 Escala ::= E Valor_entero

90 Identificador_general ::= [ * ] Identificador
    { . Identificador | [ Expresión ] | -> Identificador }

91 Identificador_de_tipo ::=
    int | char | float | bool | Identificador

92 *Identificador* **::=** *Letra* **{** *Letra* **|** *Guión* **|** *Dígito* **}**

93 *Letra* **::=**

A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z

94 *Guión* **::=** _

95 *Cadena_de_caracteres* **::=** "...caracteres normales o \escapes..."

96 *Carácter* **::=** 'carácter normal o \escape'

## A.14 Índice de reglas BNF

*Argumento*, 41
*Asignación*, 48
*Bloque*, 43
*Cabecera_función*, 38
*Cabecera_procedimiento*, 39
*Cabecera_subprograma*, 37
*Cadena_de_caracteres*, 95
*Campo*, 29
*Campo_array*, 32
*Campo_puntero*, 31
*Campos_igual_tipo*, 30
*Carácter*, 96
*Caso*, 54
*Constante_cadena*, 11
*Constante_estructurada*, 12
*Constante_simple*, 10
*Continue*, 63
*Declaración_de_bloque*, 46
*Declaración_de_constante*, 9
*Declaración_de_tipo*, 18
*Declaración_de_variable*, 33
*Declaración_global*, 7
*Declaración_interfaz*, 8
*Decremento*, 50
*Delete*, 65
*Dimensiones*, 13

*Do_while*, 58
*Dígito*, 87
*Elemento*, 84
*Escala*, 89
*Expresión*, 70
*Expresión_AND*, 72
*Expresión_OR*, 71
*Expresión_igualdad*, 73
*Expresión_numérica*, 74
*Factor*, 76
*Factor_cualificado*, 77
*For_creciente*, 59
*For_creciente_menor*, 60
*For_decreciente*, 61
*Guión*, 94
*Identificador*, 92
*Identificador_de_tipo*, 91
*Identificador_general*, 90
*If_else*, 51
*Include*, 5
*Incremento*, 49
*Inicio_estructurado*, 14
*Item*, 28
*Letra*, 93
*Lista_de_argumentos*, 40
*Lista_de_campos*, 27
*Lista_de_casos*, 53
*Lista_de_expresiones*, 69
*Lista_de_identificadores*, 21
*Lista_de_inicios*, 17
*Lista_de_items*, 26
*Lista_de_opciones*, 55
*Lista_de_valores*, 16
*Lista_de_variables*, 36
*Lista_estructurada*, 15
*Llamada_a_procedimiento*, 62
*Módulo_implementación*, 4
*Módulo_interfaz*, 3
*Opción*, 56
*Operador_AND*, 79

*Operador_OR*, 78
*Operador_comparación*, 81
*Operador_igualdad*, 80
*Operador_multiplicador*, 83
*Operador_sumador*, 82
*Pragma_once*, 6
*Programa_principal*, 2
*Return*, 64
*Secuencia_de_declaraciones*, 44
*Secuencia_de_sentencias*, 45
*Secuencia_dígitos*, 86
*Sentencia*, 47
*Sentencia_nula*, 68
*Subprograma*, 42
*Switch*, 52
*Throw*, 66
*Tipo_array*, 22
*Tipo_enum*, 20
*Tipo_puntero*, 25
*Tipo_sinónimo*, 19
*Tipo_struct*, 23
*Tipo_unión*, 24
*Try*, 67
*Término*, 75
*Unidad_de_compilación*, 1
*Valor_entero*, 85
*Valor_real*, 88
*Variable_estructurada*, 35
*Variable_simple*, 34
*While*, 57

# Apéndice B

# Manual de Estilo

Este apéndice constituye el *Manual de Estilo* básico para la realización de programas en **C±**. El objetivo de un manual de estilo es recopilar un conjunto de *buenas prácticas* de programación para establecer una forma sistemática en la elaboración de todos los programas. A lo largo del libro se han presentado algunas de las normas de programación y en este apéndice se han recopilado para facilitar la elaboración de los programas con una mayor calidad, facilitar su mantenimiento y evitar ambigüedades.

Hay que tener en cuenta que en cualquier desarrollo de software pueden intervenir decenas o incluso centenares de programadores y es absolutamente necesario disponer de un manual de estilo para que todos los que participan en el proyecto tengan un estilo de programación único, uniforme y asumible por todos. El estilo debe ser fruto de la experiencia en el desarrollo de proyectos anteriores y para que su empleo sea efectivo y generalizado es necesario disponer de las herramientas adecuadas para verificar que todos los programas siguen el estilo establecido. Así, una norma se considera **Obligatoria** cuando se debe cumplir siempre y además se dispone de una adecuada herramienta de verificación de su cumplimiento. Una norma es **Recomendable** cuando es aconseja su utilización pero la verificación sistemática no es posible en todos los casos.

Como caso especial, se han incluido como reglas de estilo algunas que son realmente restricciones de **C±** respecto a C++, y que obviamente no son comprobadas por el compilador de C++ usado para compilar los programas en **C±**.

En este apéndice se han agrupado las normas en cinco grandes apartados que se describen a continuación.

## B.1 Aspectos generales

En este apartado se agrupan las normas que se deben aplicar en cualquier punto de un programa con carácter general.

### B.1.1 Sintaxis

- **Obligatoria:** Cualquier programa debe ajustarse estrictamente a la sintaxis de **C±** que se detalla en el apéndice A.
- **Obligatoria:** En cualquier bloque de programa todas las declaraciones se deben agrupar en la parte declarativa que siempre precederá a la parte ejecutiva compuesta por una secuencia de sentencias. Por tanto, nunca se pueden mezclar declaraciones y sentencias ejecutables.

### B.1.2 Encolumnado

- **Obligatoria:** Siempre se debe utilizar el mismo indentado en el encolumnado de todos los programas de un mismo proyecto.
- **Obligatoria:** El indentado máximo será de 4 espacios en blanco y el mínimo de 2 espacios en blanco.
- **Recomendable:** Es aconsejable utilizar un indentado de 2 espacios en blanco.

### B.1.3 Comentarios

- **Obligatoria:** El programa deberá tener al menos un comentario de cabecera de programa que incluya el nombre del programa y del autor, una breve descripción de lo que hace y la fecha de su última modificación. **NOTA:** *En la mayoría de los ejemplos del libro no se incluye esta cabecera por la limitación de espacio y porque casi siempre resultaría redundante ya que la descripción se detalla en las explicaciones del texto.*
- **Obligatoria:** Se utilizará un comentario de cabecera de sección para documentar cada una de las partes importantes del programa y separarlas convenientemente.
- **Recomendable:** Es aconsejable utilizar comentarios-orden y comentarios al margen siempre que se requiera cualquier aclaración adicional sobre los refinamientos empleados o el significado de algún elemento del programa.
- **Obligatoria:** Las sentencias relativas a un comentario-orden se agruparán mediante llaves `{...}` para realizar un indentado con las sentencias del comentario-orden agrupadas.

### B.1.4 Identificadores

- **Obligatoria:** Los identificadores deben ser nombres que reflejen su utilidad sin que sea necesario ningún comentario adicional sobre su significado.
- **Obligatoria:** Nunca se utilizarán identificadores formados sólo por letras mayúsculas (salvo que se trate de siglas u otra abreviatura de uso común).
- **Obligatoria:** Los identificadores compuestos realizados por concatenación de palabras se realizarán intercalando una letra mayúscula al comenzar la nueva palabra o bien separando las palabras por el guión bajo (_).
- **Obligatoria:** Los identificadores de un tipo de dato serán siempre identificadores compuestos que comenzarán con el prefijo `Tipo`.
- **Recomendable:** Es aconsejable que los identificadores de constante y subprograma comiencen por una letra mayúscula.
- **Recomendable:** Es aconsejable que los identificadores de variable comiencen por una letra minúscula.
- **Recomendable:** Es aconsejable que los identificadores de funciones sean nombres (del valor resultante).
- **Recomendable:** Es aconsejable que los identificadores de procedimientos sean verbos o nombres de acciones.

## B.2 Declaraciones

- **Obligatoria:** Cuando las declaraciones de un bloque de programa no superen el tamaño de una página, el orden de las declaraciones será siempre: Constantes, Tipos, Variables y finalmente Subprogramas.
- **Recomendable:** Cuando las declaraciones de un bloque de programa tengan cierta complejidad y la aplicación estricta de la regla anterior pueda disminuir la claridad, es aconsejable agrupar las declaraciones por aquellos elementos significativos del programa que estén relacionadas entre sí. En este caso, aunque no existe un orden riguroso para declarar constantes, variables, tipos y subprogramas sí que es aconsejable que las declaraciones de un mismo elemento conserven entre sí el orden establecido en la regla anterior. Como se ha visto en el tema 14, los tipos abstractos de datos son una forma adecuada adecuada de agrupar todas las declaraciones relativas a un elemento significativo del programa.

### B.2.1 Constantes

- **Recomendable:** Es aconsejable agrupar todas las declaraciones de las constantes globales de un programa en un punto único para que su parametrización resulte clara y evidente.
- **Recomendable:** Es aconsejable no utilizar directamente valores constantes numéricos o literales en las sentencias ejecutables de un programa (salvo casos triviales). Resulta preferible definirlos como constantes con nombre.

### B.2.2 Tipos de datos

- **Obligatoria:** Todos los nuevos tipos de datos del programa se deben definir con un `typedef`. No se permite la existencia de tipos anónimos.
- **Obligatoria:** La declaración de un nuevo tipo sinónimo o puntero se realizará siempre en una sola línea de código.

#### B.2.2.1 Formato para tipo enumerado

- **Recomendable:** Siempre que sea posible es aconsejable que la declaración de un tipo enumerado se realice en una única línea de programa utilizando el siguiente formato:

```
typedef enum TipoNuevo {Valor, Valor, . . . , Valor}
```

- **Obligatoria:** Cuando el número de elementos del tipo enumerado sea amplio o en general resulte aconsejable utilizar más de una línea de programa en la declaración, el formato utilizado sera el siguiente:

```
typedef enum TipoNuevo {
  Valor, Valor, . . . . . Valor,
  Valor, Valor, . . . . . Valor,
  . . . .
  Valor, Valor, . . . . . Valor
}
```

#### B.2.2.2 Formato para tipo formación

- **Obligatoria:** La declaración de un tipo formación se realizará siempre en una única línea de programa utilizando el siguiente formato:

```
typedef TipoElemento TipoNuevo [Dimension][Dimension]...
```

- **Recomendable:** Es aconsejable que el valor de cada `Dimension` que fija el tamaño de la formación se declarare previamente como constante con nombre.

#### B.2.2.3 Formato para tipo registro (struct)

- **Obligatoria:** La declaración de un tipo registro (`struct`) nunca se realizará en una única línea de programa. El formato utilizado sera el siguiente:

```
typedef struct TipoNuevo {
  TipoCampo NombreCampo;
  TipoApuntado * NombreCampo;
  TipoCampo NombreCampo;
  . . . .
}
```

- **Obligatoria:** Todos los tipos `TipoCampo` son identificadores de tipo que tienen que haber sido declarados previamente.
- **Obligatoria:** Cuando un campo es de tipo puntero (*), el tipo de dato al que apunta `TipoApuntado` también tiene que haber sido declarado previamente o bien ser el propio nuevo tipo `TipoNuevo` que se está definiendo. Los campos de tipo puntero son el único caso de tipo anónimo permitido.

#### B.2.2.4 Formato para tipo registro variante (union)

- **Obligatoria:** La declaración de un tipo registro variante (`union`) nunca se realizará en una única línea de programa. El formato utilizado sera el siguiente:

```
typedef union TipoNuevo {
  TipoCampo NombreCampo;
  TipoApuntado * NombreCampo;
  TipoCampo NombreCampo;
  . . . .
}
```

- **Obligatoria:** Todos los tipos `TipoCampo` son identificadores de tipo que tienen que haber sido declarados previamente.
- **Obligatoria:** Cuando un campo es de tipo puntero (*), el tipo de dato al que apunta `TipoApuntado` también tiene que haber sido declarado

previamente o bien ser el propio nuevo tipo `TipoNuevo` que se está definiendo. Los campos de tipo puntero son el único caso de tipo anónimo permitido.

### B.2.3 Variables

- **Obligatoria:** En la declaración de variables sólo se puede utilizar un identificador de tipo válido predefinido o declarado previamente. Dicho de otra manera no se pueden declarar variables de tipo anónimo.
- **Obligatoria:** Cuando se inicializa una variable sólo está permitida la declaración de esa variable de manera individual. Por tanto, no está permitido realizar inicializaciones de ninguna variable cuando se declaran como una lista de variables del mismo tipo.
- **Obligatoria:** Cuando se realiza la asignación de un resultado de tipo distinto al tipo de la variable es obligatorio realizar siempre una conversión explícita del resultado al tipo de la variable.
- **Recomendable:** Es aconsejable realizar la inicialización de la variable en la misma declaración.

### B.2.4 Subprogramas

- **Recomendable:** Dentro del bloque de un subprograma no se deben realizar ninguna asignación a un argumento pasado por valor. Cualquier asignación a un argumento formal pasado por valor sólo tiene efecto dentro del bloque del subprograma y no se transmite al argumento real después de finalizar la ejecución del subprograma por lo que resulta confuso y se debe evitar.
- **Recomendable:** Salvo que sea imprescindible no es recomendable utilizar redefinición de elementos.
- **Recomendable:** No se debe utilizar doble referencia salvo que el subprograma se diseñe pensando en esta posibilidad.
- **Obligatoria:** La declaración de la cabecera de un subprograma se realizará en una sola línea siempre que lo permitan el número de argumentos y la longitud de los identificadores y además no se requiera ningún comentario adicional para explicar el significado de cada argumento. En este caso la cabecera se formateará como se indica en los siguientes apartados:

  **Función**

```
TipoResultado NombreFuncion (Lista de argumentos) {
  Bloque de la función
```

```
}
```

**Procedimiento**

```
void NombreProcemiento (Lista de argumentos) {
  Bloque del procedimiento
}
```

- **Recomendable:** Cuando la cabecera resulte poco clara en una única línea por el excesivo número de argumentos, la necesidad de explicar el significado de cada argumento o cualquier otro motivo, la cabecera se formateará con un solo argumento por línea según se indica en los siguientes apartados:

**Función**

```
TipoResultado NombreFuncion (
  TipoArgumento NombreArgumento,        /* paso por valor */
  TipoArgumento & NombreArgumento,      /* paso por referencia */
  const TipoArgumento NombreArgumento, /* vector por valor */
  TipoArgumento NombreArgumento,        /* vector por refer. */
  . . . .
) {
  Bloque de la función
}
```

**Procedimiento**

```
void NombreProcemiento (
  TipoArgumento NombreArgumento,        /* paso por valor */
  TipoArgumento & NombreArgumento,      /* paso por referencia */
  const TipoArgumento NombreArgumento, /* vector por valor */
  TipoArgumento NombreArgumento,        /* vector por refer. */
  . . . .
) {
  Bloque del procedimiento
}
```

## B.2.5 Tipos abstractos de datos

- **Obligatoria:** Un tipo abstracto de dato siempre se declarará mediante un tipo registro (`struct`) con funciones y/o procedimientos encapsulados dentro del `struct`. El formato de declaración será el siguiente:

```
typedef struct TipoNuevo {
  TipoCampo NombreCampo;
```

```
  TipoCampo * NombreCampo;
  . . . .
  void NombreProcedimiento (Argumentos);
  TipoResultado NombreFuncion (Argumentos);
  . . . .

private:
  TipoCampo NombreCampo;
  TipoCampo * NombreCampo;
  . . . .
  void NombreProcedimiento (Argumentos);
  TipoResultado NombreFuncion (Argumentos);
  . . . .
}
```

- **Obligatoria:** En primer lugar se agruparán todos los elementos publicos. A continuación se declararán los elementos privados precedidos de la palabra clave `private:`
- **Obligatoria:** La declaración de los campos del tipo abstracto de datos se realizará de la misma forma que cualquier otro registro.
- **Obligatoria:** Para la declaración de las funciones y/o procedimientos se utilizarán las mismas normas de los subprogramas.

## B.3 Sentencias

- **Recomendable:** Para visualizar la separación entre las declaraciones y el comienzo de las sentencias del programa es aconsejable dejar al menos una línea en blanco.
- **Recomendable:** Es aconsejable que en una misma línea de programa sólo se escriba una única sentencia acabada en punto y coma.
- **Obligatoria:** El formato de las sentencias cuya sintaxis se define en función de cualquier sentencia del lenguaje de una manera *recursiva* nunca se puede escribir en una única línea de programa. Más concretamente, el formato que se debe utilizar dependiendo del tipo de sentencia es el siguiente:

  **Selección IF**

```
if (Condicion) {
  Sentencias
} else if (Condicion) {
  Sentencias
} else if (Condicion) {
  . . . .
```

```
} else {
  Sentencias
}
```

### Selección SWITCH

```
switch ( Expresion ) {
case valor:
  Sentencias
  break;
case valor:
case valor:
  Sentencias
  break;
  . . . .
default:
  Sentencias
}
```

### Bucle WHILE

```
while (Condicion) {
  Sentencias
}
```

### Bucle DO

```
do {
  Sentencias
} while (Condicion);
```

### Bucle FOR incremental

```
for (int indice=ValorInicial; indice <= ValorFinal; indice++) {
  Sentencias
}
```

### Bucle FOR decremental

```
for (int indice=ValorInicial; indice >= ValorFinal: indice--) {
  Sentencias
}
```

### Bucle FOR incremental menor

```
for (int indice=ValorInicial; indice < ValorFinal; indice++) {
  Sentencias
}
```

### Bloque TRY-CATCH para manejo de excepciones

```
try {
  Sentencias
} catch (Tipo excepcion) {
  Sentencias
}
```

- **Obligatoria:** La variable `indice` de un bucle FOR puede ser utilizada dentro del bucle pero nunca debe ser modificada, pues se perdería el control automático de las repeticiones.

## B.4 Expresiones

- **Obligatoria:** Hay que tener en cuenta que en la evaluación de las expresiones complejas el orden por defecto que se sigue viene fijado por nivel de prioridad que tienen asignadas las distintas operaciones. Si no se utilizan paréntesis, el orden de evaluación de los 7 niveles de prioridad es el siguiente:

| | |
|---|---|
| 1. Operador Unario: | `! + - * &` |
| 2. Operador Multiplicativo: | `* / %` |
| 3. Operador Aditivo: | `+ -` |
| 4. Operador de Comparación: | `> >= < <=` |
| 5. Operadores de Igualdad: | `== !=` |
| 6. Operador AND lógico: | `&&` |
| 7. Operador OR lógico: | `\|\|` |

- **Recomendable:** Es aconsejable utilizar paréntesis adicionales para evitar cualquier ambigüedad o dificultad de interpretación de la expresión. Por ejemplo, cuando se combinen operadores de comparación y lógicos.
- **Recomendable:** No es aconsejable utilizar paréntesis adicionales en aquellas expresiones que, aprovechando los niveles de prioridad por defecto del lenguaje, estén ampliamente consensuadas y no planteen ninguna duda en su interpretación.
- **Obligatoria:** No está permitido realizar expresiones aritméticas entre operandos de distintos tipos. Siempre es obligatorio realizar una conversión explícita de tipos para precisar la operación aritmética que se quiere realizar. Esta regla trata de evitar resultados inesperados provocados por las conversiones numéricas por defecto.
- **Obligatoria:** No está permitida ninguna forma de comparación entre elementos de distintos tipos.

- **Obligatoria:** Los operadores lógicos (`&&` y `||`) sólo se pueden utilizar con elementos de tipo SI(cierto)/NO(falso).

## B.5 Punteros

- **Obligatoria:** Está prohibido utilizar los punteros como formaciones.

## B.6 Módulos

- **Obligatoria:** El fichero fuente del módulo principal debe tener el mismo nombre que el programa ejecutable final, y la extensión `.cpp`.
- **Obligatoria:** Por cada módulo no principal debe haber dos ficheros fuente, con el mismo nombre que el nombre lógico del módulo y extensiones `.h` (interfaz) y `.cpp` (implementación).
- **Obligatoria:** Cada fichero de interfaz de un módulo no principal (*modulo*`.h`) debe comenzar con la directiva `#pragma once`.
- **Obligatoria:** Cada fichero de implementación de un módulo no principal (*modulo*`.cpp`) debe incluir su propio fichero de interfaz (`#include "`*modulo*`.h"`).
- **Obligatoria:** Las constantes y tipos públicos de un módulo deben declararse solamente en el fichero de interfaz (*modulo*`.h`).
- **Obligatoria:** Las variables y subprogramas públicos deben definirse completamente en el fichero de implementación (*modulo*`.cpp`) y además deben declararse en el fichero de interfaz (*modulo*`.h`).
- **Obligatoria:** La declaración de cada variable pública en el fichero de interfaz debe ir precedida de la palabra clave `extern` y no debe incluir especificación de valor inicial.
- **Obligatoria:** La declaración de cada subprograma público en el fichero de interfaz debe consistir solamente en su cabecera (tipo, nombre y argumentos).
- **Obligatoria:** La definición de cada variable global y subprograma no públicos en el fichero de implementación de un módulo no principal debe ir precedida de la palabra clave `static`.
- **Recomendable:** Conviene que los nombres de todos los elementos públicos de un módulo comiencen por un mismo prefijo corto y con valor nemotécnico que identifique el módulo al que corresponden. Esto es importante en aplicaciones con muchos módulos o en el caso de librerías de módulos reutilizables.

# Apéndice C

# Notación lógico-matemática

En este apéndice se describe la notación utilizada en las especificaciones formales que aparecen en algunos puntos de este libro para anotar y razonar sobre la corrección de ciertos fragmentos de código. La notación empleada es esencialmente la notación matemática habitual, con ciertas simplificaciones y adaptaciones para hacerla más asequible a los lectores de este libro.

En lo que sigue, se usan los siguientes nombres para designar elementos genéricos del tipo que se indica:

$a\ b\ c$: valores numéricos
$p\ q$: valores lógicos
$e$: elemento genérico (valor de cualquier tipo)
$i\ j\ k$: índice (valor entero)
$c\ v$: colecciones (conjuntos o vectores)
$P(x...)$: predicado (expresión lógica que depende de ciertos argumentos)

## C.1 Operadores numéricos

Las operaciones aritméticas habituales:

| | | |
|---|---|---|
| $a+b$ | Suma | $a$ más $b$ |
| $a-b$ | Diferencia | $a$ menos $b$ |
| $a \times b$ | Producto | $a$ multiplicado por $b$ |
| $a/b$ | Cociente | $a$ dividido por $b$ |
| $-a$ | Cambio de signo | menos $a$ |
| $\Sigma c$ | Sumatorio | suma de los elementos de $c$ |

También se usarán funciones matemáticas de todo tipo que se supongan ya definidas: $máximo(x, y, ...)$, factorial $x!$, etc.

## C.2 Operadores de comparación

Igualdad y relación de orden:

| | | |
|---|---|---|
| $a > b$ | mayor | $a$ es estrictamente mayor que $b$ |
| $a \geq b$ | mayor o gual | $a$ es mayor o igual que $b$ |
| $a < b$ | menor | $a$ es estrictamente menor que $b$ |
| $a \leq b$ | menor o igual | $a$ es menor o igual que $b$ |
| $a = b$ | igual | $a$ es igual a $b$ |
| $a \neq b$ | diferente | $a$ es distinto de $b$ |

## C.3 Operadores lógicos

Operadores de la lógica booleana:

| | | |
|---|---|---|
| $p \wedge q$ | conjunción | $p$ y además $q$ |
| $p \vee q$ | disyunción | $p$ o $q$ o ambos |
| $\neg p$ | negación | no $p$ |
| $p \rightarrow q$ | implicación | si es cierto $p$ también es cierto $q$ |

Formas abreviadas:

| | | |
|---|---|---|
| $a < b < c < ...$<br>$a \leq b \leq c \leq ...$<br>... | conjunción | $a < b \ \wedge \ b < c \ \wedge \ c < ...$<br>(y también para otras combinaciones de operadores de comparación) |

## C.4 Colecciones

Se usa la misma notación para colecciones no ordenadas (conjuntos) y ordenadas (vectores, listas o secuencias). Cuando sea oportuno se usarán los conjuntos como vectores, y viceversa.

| | | |
|---|---|---|
| $\{e \in c, \ P(e)\}$ | comprensión | conjunto de elementos $e$ de un conjunto universal $c$, que cumplen la propiedad $P$ |
| $v[i]$ | elemento de vector | $i$-simo elemento de $v$ ($v_i$) |
| $cardinal(c)$ | cardinal | número de elementos de $c$ |

Formas abreviadas:

| | | |
|---|---|---|
| $i..j$ | rango o intervalo | colección de valores correlativos desde $i$ hasta $j$, ambos inclusive (si $i$ es mayor que $j$, entonces denota el conjunto vacío) |
| $i < k < j$<br>$i < k \leq j$<br>$i < k \leq j$<br>$i \leq k \leq j$ | rango o intervalo | conjunto de valores correlativos $k$ desde $i$ hasta $j$ (incluidos o no los extremos) |
| $\{v[i],\ P(v[i])\}$ | comprensión | colección de elementos de $v$ que cumplen la propiedad $P$ |
| $v[P(i)]$ | comprensión | colección de elementos de $v$ cuyos índices cumplen la propiedad $P$ |
| $v[c]$ | sección | colección de elementos de $v$ cuyos índices son los elementos de $c$ |

Las llaves de la expresión de un conjunto mediante $\{elemento,\ propiedad\}$ se pueden omitir si la expresión ya va entre paréntesis o corchetes.

## C.5 Cuantificadores

Universal y existencial:

| | | |
|---|---|---|
| $\forall e \in c \bullet P(e)$ | para todo | todos los elementos de $c$ cumplen la propiedad $P$ |
| $\exists e \in c \bullet P(e)$ | existe | al menos un elemento de $c$ cumple la propiedad $P$ |

## C.6 Expresiones condicionales

Expresión con formas alternativas dependiendo de alguna condición:

| | | |
|---|---|---|
| $c \Rightarrow A \vert B$ | si, en otro caso | si se cumple $c$ se evalúa $A$, en otro caso se evalúa $B$ |
| $c1 \Rightarrow E1 \vert c2 \Rightarrow E2 \vert ... \vert E$ | si, o bien si, ..., en otro caso | si se cumple $c1$ se evalúa $E1$, si se cumple $c2$ se evalúa $E2$, ..., en otro caso se evalúa $E$ |

Es equivalente a las sentencias IF-THEN-ELSE, pero para expresiones en lugar de acciones.

# Bibliografía

ACERA GARCÍA, Miguel Ángel: *C/C++*. Edición revisada y actualizada 2010. Anaya Multimedia, 2009

BALCAZAR, José Luis: *Programación metódica*. McGraw-Hill, 2001.

CEBALLOS SIERRA, Francisco Javier: *C/C++ Curso de programación*. Tercera edición. Editorial RA-MA, 2007.

DEITEL, Harvey M. y DEITEL, Paul J.: *Como programar C++*. Prentice-Hall Mexico, 2003.

GARCÍA-BERMEJO, José Rafael: *Programación estructurada en C*. Pearson Educacion. 2008

JOYANES AGUILAR, Luis: *Programación en C++. Algoritmos, estructuras de datos y objetos*. Segunda edición. McGraw-Hill, 2006.

KERNIGHAN, Brian W. y RITCHIE, Dennis M.: *El lenguaje de programación C*. Segunda edición. Prentice-Hall Iberoamericana. 1991.

PEÑA MARÍ, Ricardo: *Diseño de programas. Formalismo y abstracción*. Tercera edición. Pearson Educacion, 2005.

STROUSTRUP, Bjarne: *El lenguaje de programación C++*. Addison-Wesley Iberoamericana, 2002.

# Índice analítico


abstracción, 20, 181, 220, 239, 385, 402
  de datos, 417, 423
  funcional, 184, 186, 392, 417
acceso directo, 352
acceso secuencial, 352
acción, 20, 98, 181
  abstracta, 186
  compuesta, 20
  parametrizada, 159, 186
  primitiva, 20
acoplamiento, 405
agregado, 238
argumentos
  formales, 155, 157, 161
  reales, 157, 161
arquitectura Von Neumann, 16
array, 228, 289
aserciones, 137
autodecremento, 62, 110, 111
autoincremento, 62, 110

bases de datos relacionales, 303
biblioteca, 429
bloque de código, 154, 155, 165, 182
bucle, 100
buenas prácticas, 23, 445
búsqueda por dicotomía, 326
búsqueda secuencial, 321

cabecera de función, 154, 155
cabecera de programa, 82
cabecera de sección, 82
cabecera de subprograma, 165, 182, 412
cadena de caracteres, 31, 235
campos, 240
caracteres de control, 31
casos generales, 167
casos minimales, 167
centinela, 315, 328
*charset*, 37
claridad, 6
clase, 379
*codepoint*, 37
cohesión, 405
comentario, 46
comentarios al margen, 83
comentarios-orden, 82
compilación segura, 403, 416
compilación separada, 403
compilador, 9
complejidad algorítmica, 143
componentes, 227
comportamiento asintótico, 146
computador, 4
condiciones, 98
constante, 27, 54
  con nombre, 54, 89
  literal, 54
  local, 155
  simbólica, 54
contorno, 328
corrección, 6, 135, 315
corrección parcial, 136

corrección total, 136
cuerpo de función, 155
cursor, 352
cálculo-$\lambda$, 13
código de máquina, 7
cómputo, 3

dato, 27
dato abstracto, 392
dato complejo, 392
dato elemental, 392
declaración, 411
definición, 411
definición de tipos, 219
depuración, 135
desarrollo ascendente, 206
desarrollo descendente, 188, 392
descomposición modular, 404
diagramas de flujo, 98
diccionario, 303
directivas para el compilador, 45
discriminante, 298
doble referencia, 173

efectos secundarios, 169
eficiencia, 7, 142
eficiencia en tiempo, 143
ejecutar, 3
elección de nombres, 85
elemento
    global, 165
    local, 165
elementos privados, 408
elementos públicos, 407
encolumnado, 80
ensayo, 135
especificación, 182, 186, 402
especificación formal, 137, 183, 315
especificación informal, 183
esquema, 21, 76
    de iteración, 121
    de recorrido, 316
    de selección, 120
    secuencial, 76
    típico de operación, 315
estructuras dinámicas, 352
evaluación en cortocircuito, 103
expresión
    abstracta, 184
    aritmética, 39, 103
    condicional, 101
    constante, 56
    lógica, 102
    parametrizada, 155, 185

fichero
    de cabecera, 408
    de implementación, 408
    de interfaz, 408
    fuente, 405
flujo de control, 98
formación, 227, 289, 299
fuerza bruta, 333
función, 154, 181
    estándar, 158
    predefinida, 157
    pura, 169, 186

glifo, 37
gráficos de tortuga, 394

hardware, 4
herencia, 379

identificadores, 51
implementación, 182
ingeniería de software, 5, 23
inicializar, 60
inserción, 322
instrucción, 17
interfaz, 182, 402
intérprete, 10
invariante, 140, 141, 315, 317

iteración, 21, 99, 100, 109, 110, 299

juego de caracteres, 31

lenguaje
- de máquina, 7
- de programación, 8
- ensamblador, 21
- fuente, 9
- objeto, 9

lenguaje C±, 25
librería, 429
librerías estándar, 158

mantisa, 30
Manual de Estilo, 23, 26, 41, 46, 52, 60, 63, 68, 80, 101, 105, 111, 163, 230, 241, 261, 373, 445
matrices, 288
- orladas, 315, 330, 345

mayúsculas (en nombres), 88
metasímbolo, 27, 435
metodología de programación, 23
minúsculas (en nombres), 88
modelo abstracto de cómputo, 11
modelo de flujo de datos, 11
modelo de terminación, 213
máquina, 1
- automática, 2
- de programa almacenado, 4
- no automática, 2
- programable, 2
- virtual, 1, 4, 10

método, 379
módulo, 206, 401
- de librería, 42, 64
- estándar, 162
- principal, 407

notación BNF, 26, 435
notación UML, 422

objeto, 379
ocultación, 165, 183, 402, 403
operación, 181
- abstracta, 188, 392
- compleja, 188, 392
- de escritura, 42
- de lectura, 64
- terminal, 188, 392

operador, 33, 39
operador unario, 103
operando, 39
orden de evaluación, 104
ordenación por inserción directa, 324
orientada a objetos, 379
overflow, 123

palabras clave, 53
palabras reservadas, 53
parámetros del programa, 89
paso de argumentos
- por referencia, 162, 163
- por valor, 162, 163

plataforma, 33
postcondición, 137, 315, 316
precondición, 137, 315, 316
predicados, 226
preproceso, 414
problemas intratables, 146
procedimiento, 42, 154, 159
- estándar, 162
- puro, 187

procesador, 20
procesadores de lenguajes, 9
programa, 2
- fuente, 9
- objeto, 9
- principal, 154, 407
- robusto, 210

programación, 5
- a la defensiva, 211
- declarativa, 15

estructurada, 21, 76, 97, 255, 258, 261, 263
funcional, 11, 12
imperativa, 11, 17
lógica, 11, 15
orientada a objetos, 11
prototipo, 412
proyectos, 416
punteros, 354

realización, 182, 186, 402
recorrido de matrices, 318
recursividad, 167, 359, 393, 394, 452
red de operadores, 13
redefinición de elementos, 171, 172
reducción, 12
reescritura, 13
refactorización, 429
referencias, 354
refinamientos sucesivos, 76, 98, 101, 152
registro, 239
registros con variantes, 298
regla de producción, 27
relación, 303
representación (de un valor), 28
reutilización, 199

secuencia, 21, 98, 99, 106, 299, 352
secuencia de escape, 31
selección, 21, 99, 100, 106, 299
selección en cascada, 108
semántica, 182
sentencia, 17
sentencia de asignación, 61
seudofunciones, 158
*shadowing*, 171
signatura, 166
sintaxis, 182
software, 4
*string*, 31, 235
subproblema, 151
subprograma, 151

tabla, 303
tamaño del problema, 143
terminación, 214
tipado fuerte, 62
tipo, 28
abstracto de datos, 28, 206, 378, 392
anónimo, 230, 241
booleano, 226
`char`, 37
definido, 32
enumerado, 222
escalar, 226
estructurado, 227
`float`, 35
`int`, 33
ordinal, 222
predefinido, 32
sinónimo, 220
vector, 229
transparencia referencial, 169, 185
tratamiento de la información, 3
tupla, 227, 238, 239, 299

unidad de compilación, 405, 406
unión, 295, 299

valor, 27
carácter, 31
entero, 29
real, 30
variable, 17, 56
dinámica, 354
local, 155
variante, 137, 141, 295, 315–317
vectores abiertos, 286
visibilidad, 165