Apuntes de Informática para tercero de I. Industriales

Luciano Sánchez

Octubre de 2002

Índice general

I Introducción a la Programación 7

1. Introducción 9
1.1. Procesadores y acciones . . . 9
1.2. Definiciones . . . 11
1.2.1. Acción, estado . . . 11
1.2.2. Algoritmo, léxico . . . 11
1.2.3. Tipos de datos . . . 12
1.3. Estructura de un computador . . . 12
1.3.1. Léxico de la unidad central de proceso . . . 12
1.3.2. Organización de la memoria de un computador . . . 13
1.4. Lenguajes de programación . . . 13
1.5. Fases en el desarrollo de un programa . . . 14

2. Estructuras Básicas de Algoritmos y de Datos 17
2.1. Concepto de variable y constante . . . 17
2.2. Tipos elementales . . . 17
2.2.1. Tipo entero . . . 17
2.2.2. Tipo real . . . 18
2.2.3. Tipo booleano . . . 19
2.2.4. Tipo carácter . . . 19
2.2.5. Tipo complejo . . . 19
2.3. Codificación de los valores de los tipos simples . . . 20
2.3.1. Codificación de un entero . . . 20
2.3.2. Codificación de un real . . . 20
2.3.3. Codificación de un carácter . . . 21
2.3.4. Codificación de un booleano . . . 21
2.3.5. Codificación de un complejo . . . 21
2.4. Reglas para la construcción de un nombre en C++ . . . 21
2.5. Declaraciones y definiciones en C++ . . . 22
2.5.1. Definición de variables y constantes . . . 22
2.6. Expresiones . . . 23
2.6.1. Expresiones de tipo entero . . . 23
2.6.2. Expresiones de tipo real y complejo . . . 23
2.6.3. Expresiones de tipo booleano . . . 24
2.6.4. Expresiones de tipo carácter . . . 24
2.7. Acción de asignación . . . 24

2.8. Acciones de entrada y salida de datos 25
2.8.1. Entrada o lectura 25
2.8.2. Salida o escritura 25
2.9. Composiciones de acciones 26
2.9.1. Composición secuencial. Bloques. 26
2.9.2. Composición condicional o alternativa 27
2.9.3. Composición iterativa 30
2.10. Comentarios 33
2.11. El tipo vector 34
2.11.1. Recorrido de los elementos de un vector 35

3. Definición de Acciones no Primitivas 37
3.1. Estructura de un algoritmo 37
3.2. Ambito de una declaración 38
3.3. Diseño descendente 41
3.4. Acciones con nombre 42
3.5. Funciones y operadores 45
3.5.1. Declaración y definición de funciones 45
3.5.2. Operadores 46
3.5.3. Sobrecarga 47
3.6. Sinónimos, referencias y punteros 47
3.7. Tiempo de vida de una variable 49
3.8. Mecanismos del paso de argumentos 50
3.9. Semántica del paso de argumentos 51
3.10. Argumentos procedurales 53
3.10.1. Modularidad y ocultación de información 54
3.11. Recursividad 55

4. Definición de Nuevos Tipos de Datos 57
4.1. Definiciones de Tipos 57
4.2. Representación de nuevos tipos de datos 57
4.2.1. Definiciones de sinónimos: Orden “typedef” 58
4.2.2. Tipos enumerados: Tipo “enum” 58
4.2.3. Tuplas: orden “struct” 59
4.3. Tipos Abstractos de Datos 61
4.3.1. Definiciones 62
4.4. Implementación de tipos abstractos de datos: Orden “class” . . . 62
4.4.1. Funciones miembro 63
4.4.2. Miembros públicos y privados 67
4.5. Extensión temporal de la definición de una variable 76
4.6. Constructores de tipos 76
4.7. Destructores 78
4.8. Asignación y paso por valor de objetos como argumentos 79
4.9. Tipos parametrizados: Orden “template” 81
4.9.1. Acciones genéricas 81
4.10. Objetos como miembros de otras clases 84

5. Estructuras de datos **87**
5.1. El tipo abstracto de datos “vector” 87
5.1.1. Representación en posiciones contiguas 87
5.1.2. Representación de matrices dispersas 88
5.1.3. Vectores de librería STL 88
5.2. El tipo abstracto de datos “pila” 89
5.3. El tipo abstracto de datos “cola” 90
5.4. El tipo abstracto de datos “cadena de caracteres” 91
5.5. El tipo abstracto de datos “arbol binario” 93
5.6. El tipo abstracto de datos “grafo” 97
5.6.1. Definiciones 98
5.6.2. Recorridos de un grafo 98
5.7. El tipo abstracto “fichero secuencial” 103
5.7.1. Clase ofstream 103
5.7.2. Clase ifstream 104
5.7.3. Entrada y salida con formato 104

6. Jerarquías de Tipos **107**
6.1. Subtipado. Herencia y Jerarquías de tipos 107
6.1.1. Subtipado en C++ 108
6.2. Acciones virtuales y tipos polimórficos. Prog. orientada al objeto . 109
6.2.1. Acciones virtuales 109
6.2.2. Tipos polimórficos. Programación orientada al objeto . . . 109
6.2.3. Tipos polimórficos en C++ 110
6.2.4. Ejemplo 110

Parte I

Introducción a la Programación

1

Introducción

1.1. Procesadores y acciones

Las descripciones de muchas tareas consisten en una secuencia de acciones. En estas tareas pueden distinguirse tres elementos:

- Un **procesador**, que es una entidad capaz de entender un enunciado y de ejecutar el trabajo indicado.
- Un **entorno**, que es el conjunto de los materiales necesarios para la ejecución del trabajo.
- Una lista de **acciones**, que son sucesos que modifican el entorno.

Un computador es un procesador diseñado para ejecutar listas de acciones relacionadas con el cálculo. El entorno sobre el que operan las acciones válidas en un computador se compone de valores a los que llamaremos **datos** y el resultado del trabajo también es un conjunto de valores.

Con un computador pueden resolverse tareas como la enunciada en este problema de Física:

Hallar un vector de módulo 3 y que sea paralelo al vector suma de los:

$$\vec{\mathbf{a}} = \vec{\mathbf{i}} + 2\vec{\mathbf{j}} + \vec{\mathbf{k}}\ ;\ \vec{\mathbf{b}} = 2\vec{\mathbf{i}} - \vec{\mathbf{j}} + \vec{\mathbf{k}}\ ;\ \vec{\mathbf{c}} = \vec{\mathbf{i}} - \vec{\mathbf{j}} + 2\vec{\mathbf{k}}$$

Solución: El vector suma de los tres dados, será:

$$\vec{s} = (1+2+1)\vec{i} + (2-1-1)\vec{j} + (1+1+2)\vec{k} = 4\vec{i} + 4\vec{k}$$

y su módulo $s = \sqrt{16+16} = \sqrt{32} = 4\sqrt{2}$
Los cosenos directores del vector $\vec{s}$ son:

$$\cos\alpha = \frac{s_1}{s} = \frac{4}{4\sqrt{2}};\ \cos\beta = \frac{s_1}{s} = 0;\ \cos\gamma = \frac{s_3}{s} = \frac{4}{4\sqrt{2}} = \frac{1}{\sqrt{2}}$$

Todo vector paralelo al hallado tiene los mismos cosenos directores, luego

$$\vec{r} = 3\cos\alpha\,\vec{i} + 3\cos\beta\,\vec{j} + 3\cos\gamma\,\vec{k} = \frac{3}{\sqrt{2}}\vec{i} + \frac{3}{\sqrt{2}}\vec{k} = \frac{3\sqrt{2}}{2}(\vec{i} + \vec{k})$$

Analizaremos la secuencia de de acciones necesaria para hallar un vector de módulo m que sea paralelo al vector suma de los vectores

$$\vec{\mathbf{a}} = \mathbf{a}_1\vec{\mathbf{i}} + \mathbf{a}_2\vec{\mathbf{j}} + \mathbf{a}_3\vec{\mathbf{k}}\ ;\ \vec{\mathbf{b}} = \mathbf{b}_1\vec{\mathbf{i}} + \mathbf{b}_2\vec{\mathbf{j}} + \mathbf{b}_3\vec{\mathbf{k}}\ ;\ \vec{\mathbf{c}} = \mathbf{c}_1\vec{\mathbf{i}} + \mathbf{c}_2\vec{\mathbf{j}} + \mathbf{c}_3\vec{\mathbf{k}}.$$

Estas acciones serían:

1. Calcular $\vec{s} = \vec{a} + \vec{b} + \vec{c}$
2. Calcular los cosenos directores de $\vec{s}$
3. Multiplicar m por los cosenos directores

y el entorno estaría compuesto por los vectores $\vec{a}, \vec{b}, \vec{c}, \vec{s}, \vec{r}$ y los valores de m, $\cos\alpha, \cos\beta$ y $\cos\gamma$. Si el procesador es capaz de sumar vectores, calcular sus cosenos directores y multiplicar vectores por escalares, la secuencia de órdenes es correcta, porque todas las órdenes pertenecen al **léxico** del procesador. Si el procesador solamente conoce cómo sumar y multiplicar escalares y calcular la raíz cuadrada y las funciones trigonométricas, será necesario aportar más detalles y *refinar* la lista de órdenes, descomponiéndolas en acciones más sencillas:

1. Calcular $\vec{s} = \vec{a} + \vec{b} + \vec{c}$
   a) $s_1 = a_1 + b_1 + c_1$
   b) $s_2 = a_2 + b_2 + c_2$
   c) $s_3 = a_3 + b_3 + c_3$
2. Calcular los cosenos directores de $\vec{s}$
   a) Calcular el módulo de $\vec{s}$: $|\vec{s}| = \sqrt{s_1^2 + s_2^2 + s_3^2}$
   b) $\cos\alpha = \frac{s_1}{|\vec{s}|}$
   c) $\cos\beta = \frac{s_2}{|\vec{s}|}$
   d) $\cos\gamma = \frac{s_3}{|\vec{s}|}$
3. Multiplicar m por los cosenos directores
   a) $r_1 = m\cos\alpha$
   b) $r_2 = m\cos\beta$
   c) $r_3 = m\cos\gamma$

Para un procesador dado, una acción es **primitiva** si el enunciado de dicha acción es suficiente para poder ejecutarla sin información suplementaria. Una acción no primitiva debe ser **descompuesta** en acciones primitivas.

Los **programas** de los computadores están escritos en lenguajes formales. Cada lenguaje contiene la definicion de su propia lista de acciones primitivas y de ciertos mecanismos para controlar la secuenciación de estas acciones. Programar consiste en escribir la solución de un problema empleando solamente las acciones definidas en el lenguaje elegido.

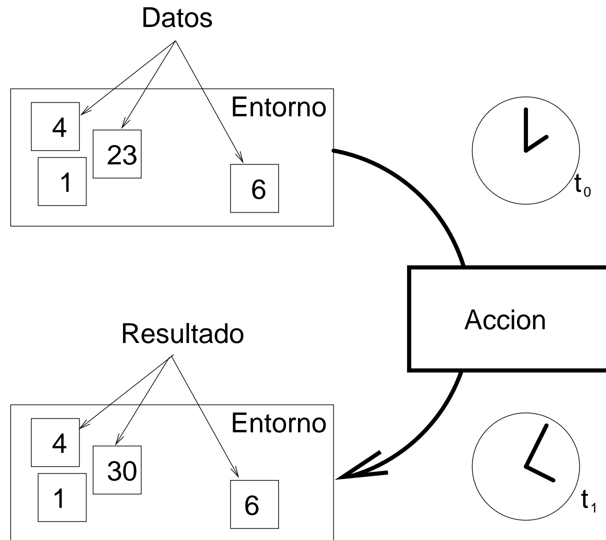

Figura 1.1: Una acción tiene lugar en un período de tiempo finito y produce un resultado previsto. Una acción modifica su entorno, que está compuesto de indicadores.

1.2. Definiciones

1.2.1. Acción, estado

Una **acción** es un acontecimiento producido por un actor, que la ejecuta. Una acción tiene lugar en un período de tiempo finito y produce un resultado bien definido y previsto, y el hecho de que dure un tiempo finito significa que se puede encontrar un instante t_0 de inicio de la acción y un instante final de la acción t_1. Por ejemplo: "Calcular $\vec{s} = \vec{a} + \vec{b} + \vec{c}$ " es una acción.

Una acción modifica su entorno. El entorno de una acción está compuesto por **indicadores** que pueden tomar distintos valores. Cada indicador tiene un nombre que permite referenciarlo. El conjunto de valores de los diferentes indicadores observados en un instante dado t del desarrollo del acontecimiento se denomina **estado** del proceso en el instante t.

El estado del sistema en el instante t_0 define los **datos** de la acción. El **resultado** es el estado del proceso en el instante t_1.

1.2.2. Algoritmo, léxico

Un procesador tiene un repertorio de acciones primitivas y de informaciones elementales identificadas. Este repertorio es el **léxico** del procesador.

Un **algoritmo** de un proceso es el enunciado de una lista de acciones primitivas que debería realizar el procesador para que, partiendo de un estado inicial, se llegue al estado final que se desea alcanzar.

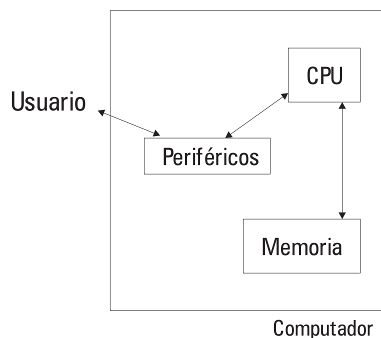

Figura 1.2: Un computador consta de tres partes: la unidad central de proceso, la memoria y los periféricos. La unidad central de proceso obtiene la lista de acciones y los valores de los indicadores de la memoria. Los periféricos son dispositivos manejados por la unidad central de proceso capaces de introducir y mostrar datos.

1.2.3. Tipos de datos

Convendremos de ahora en adelante en que cada indicador pertenece a un tipo de datos determinado. Usaremos la palabra "**tipo**" como sinónima de **conjunto**. Por ejemplo, 7 es del tipo "entero" ($7 \in \mathbf{Z}$), 'martes' es del tipo "día de la semana" ('martes'$\in$ {lunes, martes, miércoles, jueves, viernes, sábado, domingo}) y (2,5, 3,0) es del tipo "par ordenado de 2 números reales" ($(2,3) \in \mathbf{R}^2$).

1.3. Estructura de un computador

Un computador está compuesto por tres bloques (ver figura 1.2):

1. La **unidad central de proceso** (CPU), que interpreta y ejecuta las acciones.
2. La **memoria**, donde se almacenan la lista de acciones y los valores de los indicadores.
3. Los **periféricos**, que permiten introducir y extraer datos en y desde la memoria del computador. Por ejemplo, el teclado y la pantalla del ordenador son periféricos.

1.3.1. Léxico de la unidad central de proceso

El léxico de la unidad central de proceso es muy sencillo. En general, consta de varias decenas de instrucciones y contiene órdenes para

1. realizar operaciones aritméticas, como la suma, la resta o el producto
2. realizar operaciones lógicas, como decidir si un número es mayor de otro
3. decidir si una operación se realiza o no, en función del valor de un indicador
4. repetir varias veces una acción, también en función del valor de un indicador

5. enviar un dato hacia un periférico o solicitar la introducción de un dato desde un periférico

1.3.2. Organización de la memoria de un computador

Tanto la lista de acciones como los datos que éstas manejan se almacenan en la memoria. La memoria de un computador es una tabla compuesta por celdas, en cada una de las cuales se puede almacenar un número entero.

El que la memoria solamente sea capaz de almacenar números no supone una limitación importante. Cada orden del léxico del procesador puede asociarse a un número, con lo que cada programa puede asociarse a una lista de números. Los valores de los indicadores, sean del tipo que sean (números reales, caracteres, imágenes, etc.) se pueden introducir de forma similar. Por ejemplo, la palabra "dato" podría almacenarse como la lista (4,1,20,15) si conviniésemos en que la letra 'a' se codifica con el número 1, la 'b' con el 2, y así sucesivamente. Veremos en el capítulo siguiente cómo se almacenan los valores de los tipos más frecuentes.

Cada una de las celdas de memoria de un computador está construida a partir de un número fijo de elementos y, a su vez, cada uno de estos elementos puede estar en uno de dos estados: encendido o apagado. Si asociamos al estado "encendido" el número 1 y al estado "apagado" el número 0, el contenido de una celda puede describirse mediante un **número binario**, que es una cadena compuesta por tantos dígitos 0 ó 1 como elementos componen la celda. Por ejemplo, si una celda del computador tuviese 32 elementos, su contenido podría ser el número binario

00000000000000000000000000011001

Con este número estamos indicando que el primer elemento (comenzando por la derecha) está encendido, el segundo y el tercero apagados, el cuarto y el quinto encendidos y los restantes 28 apagados. Por convenio, calcularemos el número almacenado en la celda de acuerdo con los siguientes criterios:

1. Cada uno de los elementos tiene un valor asociado. El número que se busca se corresponde con la suma de los valores asociados a todos los elementos que están en estado "encendido".

2. El valor del primer elemento es 1, y el valor de cada elemento del segundo en adelante es el doble del valor del elemento que lo precede.

En el caso anterior, el valor almacenado es $1 + 8 + 16 = 25$.

Usaremos la palabra **bit** (abreviatura de **bi**nary digi**t**) para referirnos al estado de cada uno de los elementos. Una lista de 8 bits (como 00001010) se denomina **byte**. El número de celdas de memoria del ordenador se mide en millones de bytes o megabytes.

1.4. Lenguajes de programación

Es bastante difícil diseñar algoritmos en el léxico de la unidad central de proceso. Por esta razón, es habitual que el programador construya sus algoritmos en

un lenguaje con un léxico más amplio y que después convierta cada una de sus acciones a un conjunto de órdenes más sencillas. Este último paso no se realiza a mano, sino automáticamente, mediante otro programa de ordenador llamado **compilador**. En este curso estudiaremos cómo construir algoritmos en el lenguaje de programación C++.

1.5. Fases en el desarrollo de un programa

La creación de un programa está dividida en cinco fases: Especificación de requisitos, diseño, análisis, codificación y verificación.

- **Especificación de requisitos**. Una especificación es una *descripción detallada del trabajo a realizar*. Las especificaciones no pueden ser ambiguas. Una especificación puede ser un texto en el que se detalle de forma clara la tarea que se desea que realice el programa. Por ejemplo: "Determinar la secuencia de acciones necesaria para hallar un vector de módulo m que sea paralelo al vector suma de los vectores

  $$\vec{\mathbf{a}} = \mathbf{a}_1\vec{\mathbf{i}} + \mathbf{a}_2\vec{\mathbf{j}} + \mathbf{a}_3\vec{\mathbf{k}}\ ;\ \vec{\mathbf{b}} = \mathbf{b}_1\vec{\mathbf{i}} + \mathbf{b}_2\vec{\mathbf{j}} + \mathbf{b}_3\vec{\mathbf{k}}\ ;\ \vec{\mathbf{c}} = \mathbf{c}_1\vec{\mathbf{i}} + \mathbf{c}_2\vec{\mathbf{j}} + \mathbf{c}_3\vec{\mathbf{k}}.$$

  donde los valores $a_1, \ldots, c_3$ son números reales y m es mayor que cero". En esta descripción se han incluido dos partes: (a) la finalidad del proceso y (b) los tipos de los datos que procesará el algoritmo y los de los datos que se obtendrán como resultado.

- **Diseño.** El diseño consiste en construir un algoritmo que solucione el problema que se ha especificado. La construcción de un algoritmo consiste en seleccionar varias acciones elementales y en organizarlas en el tiempo para obtener el resultado correspondiente a las acciones acumuladas.

  Las acciones elementales que se emplean en el diseño no han de coincidir necesariamente con las definidas en ningún lenguaje de programación de computadores, aunque deben corresponderse con composiciones de estas últimas.

- **Análisis**. El análisis de un algoritmo consiste en estudiar lo eficaz que éste es resolviendo el problema. En general existen muchos algoritmos distintos que resuelven el mismo problema, pero no todos ellos tardan el mismo tiempo.

- **Codificación**. Una vez que el algoritmo está diseñado, se convierte al lenguaje de programación que se desee. En esta etapa se elige una representación para los datos que el programa manejará y se reemplazan las acciones del algoritmo por acciones primitivas válidas en el lenguaje de programación.

- **Verificación**. La verificación consta de tres aspectos: la demostración de la corrección del algoritmo o *verificación formal*, la prueba del programa (en inglés *software testing*) y la depuración.

  La demostración de la corrección consiste en *demostrar* matemáticamente que el programa es correcto. Puede hacerse antes de codificar el algoritmo en un lenguaje de programación, porque no requiere el uso del computador.

La prueba del programa consiste en elegir un conjunto representativo de datos, ejecutar el programa con ellos y comprobar que los resultados son los esperados. El que un programa supere con éxito su prueba no indica que éste sea correcto, salvo si el conjunto de entrada contiene todos los casos posibles. Aún así, la prueba se realiza cuando el programa tiene un tamaño para el que la demostración de la corrección es impracticable. El que un programa no supere una prueba sí implica, obviamente, que es incorrecto.

La depuración consiste en detectar los errores cometidos en la fase de codificación mediante la ejecución del programa.

2

Estructuras Básicas de Algoritmos y de Datos

2.1. Concepto de variable y constante

Para que un indicador pueda ser manipulado por el computador, este último ha de almacenar una representación del valor del indicador en su memoria. El espacio de memoria que ocupa la representación del valor de un indicador se denomina **objeto de almacenamiento** o simplemente **objeto**. Una **constante** es un objeto cuyo valor no puede ser cambiado. Una **variable** es un objeto cuyo valor puede cambiar con el tiempo (ver figura 2.1). Cada objeto tiene uno o más **nombres** (palabras con las que se hace referencia al valor del objeto) y un **tipo de datos**, que es el conjunto de los valores del indicador que el objeto puede almacenar.

2.2. Tipos elementales

Los tipos elementales comprenden los números enteros, los valores lógicos (verdadero, falso), un conjunto de caracteres de escritura, los números reales y los números complejos.

2.2.1. Tipo entero

El tipo **entero** comprende un subconjunto de los enteros cuyo tamaño puede variar según el computador y el lenguaje de programación que se emplee. Se supone, en cualquier caso, que todas las operaciones con datos de este tipo son exactas, y que el proceso de cálculo se interrumpe cuando aparece un resultado que está fuera del subconjunto de números representado en el computador. Los operadores definidos sobre los enteros son la suma (+), la resta (−), la multiplicación (∗), la división (/) y el módulo (%). La división de números enteros produce

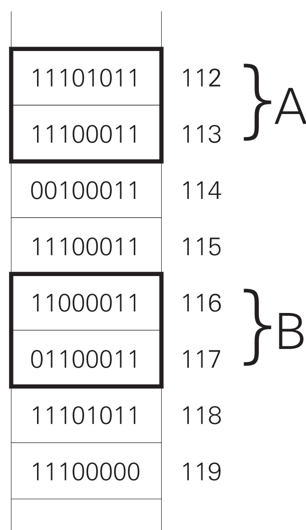

Figura 2.1: Un objeto tiene un nombre y ocupa un espacio, en el que se almacena un valor. Los objetos son variables o constantes. El valor de una constante no puede variar con el tiempo y el de una variable sí. Los distintos valores que una variable toma en el tiempo siempre pertenecen al mismo tipo de datos.

un resultado entero, y el operador de módulo (%) produce el resto de dividir un número entero entre otro:

$$(m \ / \ n) * n + (m \ \% \ n) = m$$

Existen varias clases de enteros, diferenciados por el rango de valores que pueden representar. Las operaciones son tanto más lentas cuanto mayor es este rango, por lo que se usa siempre el entero más corto posible. Los rangos de cada clase de enteros y las palabras empleadas en C++ para definirlos son:

1. `short int`: entre -32768 y 32767
2. `int`: entre -2147483648 y 2147483647
3. `unsigned short int`: entre 0 y 65535
4. `unsigned int`: entre 0 y 4294967295

En algunos computadores existen además los tipos `long int` y `unsigned long int`, con un rango de valores mayor que `int` y `unsigned int`, respectivamente. En los demás casos los tipos `long int` e `int` coindicen. Observe que los tipos `unsigned` no pueden representar números negativos.

2.2.2. Tipo real

El tipo **real** designa un subconjunto de los números reales. Al contrario que en la aritmética de los números enteros, en la aritmética de tipo real se permite una cierta imprecisión, dentro de los límites de error de redondeo producido por

el cálculo con un número finito de dígitos. Pueden aplicarse todos los operadores vistos para los números enteros, excepto el módulo (%).

Existen tres clases de números reales en C++. El tipo `float` designa un número real en el rango -10^{38} y 10^{38} y con una precisión de 7 cifras significativas. `double` designa un número real en el rango -10^{308} y 10^{308} y con una precisión de 15 cifras significativas. El tipo `long double` designa a un número real en el rango el rango -10^{4932} y 10^{4932} y con una precisión de 19 cifras significativas.

Las operaciones entre `float` son más rápidas que las operaciones entre `double`, que a su vez son más rápidas que las operaciones con `long double`.

2.2.3. Tipo booleano

El tipo **booleano** se declara con la palabra `boolean` y tiene dos valoes, **true** y **false**. Los operadores booleanos son la conjunción ("y"), disyunción ("o") y la negación ("no"), que se representan mediante los operadores `&&`, `||` y `!`. Por ejemplo, `true && false` es `false`, `true || false` es `true` y `!true` es `false`.

La certeza de una afirmación es un valor booleano. Existen operadores con argumentos de tipo entero, real, carácter o complejo que producen un valor booleano. Estos son: menor que (<), mayor que (>), menor o igual que (<=), mayor o igual que (>=), igual a (==) y distinto de (!=). Por ejemplo, $2 < 3$ es `true` y $(2 < 3)\&\&(3 < 2)$ es `false`.

2.2.4. Tipo carácter

El tipo **carácter** comprende el conjunto de caracteres utilizado por el computador. El más empleado (no todos los computadores emplean el mismo juego de caracteres) se denomina código ASCII (American Standard Code for Information Interchange). Consta de 95 caracteres de impresión y 33 caracteres de control.

Los subconjuntos de letras y números están ordenados. Por ejemplo, `'A'<= 'J'` es `true` y `'1' > '0'` es `false`. En el código ASCII las mayúsculas preceden a las minúsculas: `'a'<'B'` es `false`.

2.2.5. Tipo complejo

Un complejo está compuesto por una pareja de números reales. Hay tres clases de números complejos en C++, y se denominan `float_complex`, `double_complex` y `long_double_complex`.

Las operaciones aritméticas definidas para los números reales también pueden emplearse con los complejos. El número $a + bi$ se denota `float_complex(a,b)`, (o bien `double_complex(a,b)` o `long_double_complex(a,b)`). Por ejemplo, `sqrt(float_complex(-1,0))` es igual a `float_complex(0,1)`. `sqrt(-1)` es un error.

2.3. Codificación de los valores de los tipos simples

2.3.1. Codificación de un entero

Un `short int` se almacena en 2 bytes, y un `int` en cuatro. La codificación del `unsigned int` y `unsigned short int` es la misma que la vista en la sección 1.3.2: el entero codificado en esos bytes se corresponde con la suma de los valores asignados a todos los bits iguales a 1, donde el valor del bit más a la derecha es 1 y cada bit vale el doble que el anterior. El menor `unsigned short int` es el $0000000000000000 = 0$, y el mayor es el $1111111111111111 = 2^{16} - 1$. El mayor `unsigned int` es el $2^{32} - 1$.

La codificación de `unsigned int` y de `int` es diferente, porque en el último tipo hay números negativos. En los enteros con signo se emplea la llamada "codificación en complemento a dos", donde

- El mayor número positivo en un `short int` es el $0111111111111111 = 2^{15} - 1$, y en un `int` es $2^{31} - 1$

- Para codificar un número negativo se invierte bit a bit su valor absoluto y se suma 1 al resultado. Por ejemplo, el `short int` de valor -27 se codifica de la siguiente forma:

  $$27 = 16 + 8 + 2 + 1 = 0000000000011011$$

  $$-27 = 1111111111100100 + 1 = 1111111111100101$$

  El menor `short int` es el -2^{15} y el menor `int` es el -2^{32}. El bit situado más a la izquierda en un número negativo es 1, y si es positivo 0.

La codificación en complemento a dos es útil porque las operaciones aritméticas entre números con signo se realizan de la misma manera que las operaciones entre números sin signo. Por ejemplo,

$$-27 + 16 = 1111111111100101 + 0000000000010000 =$$

$$1111111111110101 = -(0000000000001010 + 1) =$$

$$-(0000000000001011) = -11$$

2.3.2. Codificación de un real

Existen varias formas de codificar un número real. En la más sencilla, llamada de *coma fija* se emplea un punto decimal y se utilizan reglas similares a las vistas para los enteros:

1. El primer bit a la izquierda del punto decimal tiene valor uno

2. Cada bit tiene el doble de valor que el bit que tiene a su derecha

Por ejemplo,

$$000000101{,}101 = 4 + 1 + 0{,}5 + 0{,}125 = 5{,}625$$

La colocación del punto decimal define el rango y la precisión del tipo. En general es más conveniente indicar con un número la posición de la coma, como se hace en la representación en *coma flotante*. En esta representación se transforma la representación en coma fija del número en una expresión de la forma $1.xxxx \cdot 2^{yy}$ y se concatenan las listas de bits $xxxx$ e yy. La lista $xxxx$ se llama *característica* y la yy *mantisa*, y ocupan un número fijo de bits. Por ejemplo, si se emplean 9 bits para la característica y 6 para la mantisa, el número anterior se codificaría como

$$101{,}101 = 1{,}01101 \cdot 2^2 = 011010000000010$$

En general se codifica la mantisa en complemento a dos para permitir exponentes negativos y se añade un bit al conjunto que es cero si el número es positivo y uno si es negativo. Por ejemplo, $-0{,}21875 = -0{,}00111_{(2} = -1{,}11 \cdot 2^{-2} =$ 1110000000111110. El bit más a la izquierda es 1 porque el número es negativo, los siguientes 9 son 110000000 y el exponente es $-(000001 + 1) = -2$.

Un float ocupa 32 bits. Un double ocupa 64 y un long double 96.

2.3.3. Codificación de un carácter

Cada carácter se codifica mediante un número. La tabla ASCII extendida permite codificar 256 caracteres distintos, que incluyen los dígitos, las letras mayúsculas, las letras minúsculas y los signos de puntuación. Las letras están ordenadas alfabéticamente, y el código de una letra mayúscula es un número menor que el código de una letra minúscula. Un carácter ocupa un byte.

2.3.4. Codificación de un booleano

Para codificar un booleano se emplea un bit. El valor 0 se asocia con el valor `false` y el 1 con el `true`.

2.3.5. Codificación de un complejo

Un número complejo se codifica concatenando las listas de bits con que se codifican sus partes real e imaginaria.

2.4. Reglas para la construcción de un nombre en C++

El nombre (también llamado **identificador**) por el que nos referimos a una variable, constante, función, tipo o clase en C++ consiste en una letra seguida de una secuencia de letras y dígitos. El subrayado (_) se considera una letra.

Las palabras que forman parte de la definición del lenguaje están reservadas y no pueden usarse como identificadores. Por ejemplo, una variable no puede llamarse `int`.

2.5. Declaraciones y definiciones en C++

Antes de que un objeto pueda ser utilizado en un programa C++ debe ser declarado. Debe declararse el tipo de todos los nombres de forma que el compilador sepa a qué entidad se refiere ese nombre.

La mayoría de las declaraciones son también definiciones; esto es, también definen la entidad a la que se refiere el nombre. Se verá más adelante que las declaraciones que no son definiciones (ver sección 3.5) indican el tipo de un objeto o de una función que se definirá en otro punto del programa.

Un nombre puede ser declarado varias veces siempre y cuando todas las declaraciones coincidan en su tipo. Por el contrario, solo puede ser definido una vez.

Una declaración consta de cuatro partes: un especificador (opcional), el nombre de un tipo, un nombre y un valor inicial (opcional). Todas las definiciones, excepto las de las funciones (sección 3.5.1) y las de los módulos (sección 3.10.1) terminan en un punto y coma.

2.5.1. Definición de variables y constantes

Las variables se definen de la forma siguiente:

`especificador tipo nombre = valor inicial;`

La sintaxis mínima es

`tipo nombre;`

donde `nombre` es el nombre de la variable y `tipo` el nombre de su tipo. Por ejemplo,

```
int a;
float b;
```

Las constantes se definen como las variables, precedidas del especificador `const` y seguidas obligatoriamente de su valor:

`const tipo c = e;`

donde `c` es el nombre de la constante y `e` su valor. Por ejemplo,

`const float PI = 3,1416;`

Pueden declararse varias variables del mismo tipo separando sus nombres por comas:

```
int a,b=20,c;
```

donde no se indican los valores iniciales de `a` y `c`, y el valor inicial de `b` es 20.

2.6. Expresiones

Una expresión se define inductivamente como

- Una constante
- Una variable
- Una función de una expresión
- Dos expresiones relacionadas con un operador

por ejemplo, si `x` es el nombre de una variable y `pi` es el nombre de una constante, las siguientes son expresiones:

$$\mathrm{pi}$$

$$0{,}34$$

$$x$$

$$\cos(x)$$

$$\mathrm{pi} + \cos(x)$$

$$0{,}34 * \sin(\mathrm{pi} + \cos(x))$$

Cada expresión es de un tipo: hay expresiones enteras, reales, lógicas, de carácter y complejas. La *evaluación* consiste en reemplazar los nombres de las variables y de las constantes por sus valores y en realizar el cálculo indicado.

2.6.1. Expresiones de tipo entero

Los operadores aritméticos son los de las cuatro operaciones básicas de suma $(+)$, resta $(-)$, multiplicación $(*)$ y división $(/)$. La división entre dos argumentos enteros produce un resultado entero. Si al menos uno de los argumentos es real, el resultado es un número real. Por ejemplo, en C++, 3/4=0. El operador de resto o módulo es el tanto por ciento (%). Por ejemplo, 7 %4=3.

2.6.2. Expresiones de tipo real y complejo

Las expresiones de tipo real y complejo pueden contener los operadores de suma, resta, multiplicación y división, junto con las funciones siguientes:

Expresión matemática	Expresión C++	Descripción
$x = \sqrt{y}$	x=sqrt(Y)	Raíz cuadrada
$x = \|y\|$	x=fabs(y)	Valor absoluto
$x = e^y$	x=exp(y)	Exponencial
$x = \log(y)$	x=log(y)	Logaritmo neperiano
$x = \log_{10}(y)$	x=log10(y)	Logaritmo decimal
$x = \sin(y)$	x=sin(y)	Seno
$x = \cos(y)$	x=cos(y)	Coseno
$x = \tan(y)$	x=tan(y)	Tangente
$x = \text{asin}(y)$	x=asin(y)	Arcoseno
$x = \text{acos}(y)$	x=acos(y)	Arcocoseno
$x = \text{atan}(y)$	x=atan(y)	Arcotangente
$x = \sinh(y)$	x=sinh(y)	Seno hiperbólico
$x = \cosh(y)$	x=cosh(y)	Coseno hiperbólico
$x = \tanh(y)$	x=tanh(y)	Tangente hiperbólica

2.6.3. Expresiones de tipo booleano

Los operadores booleanos son && ($\wedge$), || ($\vee$), ! ($\neg$). Las comparaciones (que devuelven un valor booleano) son == (igual), > (mayor que), < (menor que), >= (mayor o igual), <= (menor o igual) y != (distinto).

2.6.4. Expresiones de tipo carácter

Una constante de tipo carácter se encierra entre comillas simples; la expresión `A` tiene el tipo del objeto `A` y se refiere al valor de una constante o variable llamada `A`. La expresión `'A'` es de tipo carácter y su valor es la letra A.

2.7. Acción de asignación

La acción de asignación da un valor a una variable. Se empleará la notación

`V = E;`

en la que

- `V` es el nombre de la variable a la que el procesador debe atribuir el valor
- = caracteriza a la acción de asignación
- `E` es una expresión del mismo tipo que `V` o de un tipo que esté contenido en el tipo de `V`. Por ejemplo, `E` puede ser de tipo entero y `V` de tipo real.

La acción de asignación se desarrolla en dos fases:

1. Se evalúa el valor de E
2. Se almacena en la dirección de memoria asociada a V el resultado de la evaluación.

Por ejemplo, si la variable V tiene el valor 2, la acción de asignación $V = V + 1$; da a V el valor 3.

La acción `x++;` es equivalente a `x=x+1;`. La acción `x--;` es equivalente a `x=x-1;`. Las acciones `a+=x`, `a-=x`, `a*=x`, `a/=x` son equivalentes a `a=a+x`, `a=a-x`, `a=a*x` y `a=a/x`, respectivamente.

2.8. Acciones de entrada y salida de datos

2.8.1. Entrada o lectura

Puede ocurrir que un valor a tratar no forme parte del entorno de un trabajo y que sea necesario introducirlo en él. Llamamos **lectura** o **entrada** de datos a una acción en que un valor se introduce en el entorno del procesador por medio de un aparato (por ejemplo, el teclado del computador). Una lectura es una asignación: toma el valor y lo asocia a una de las variables del entorno. La lectura del valor de la variable `v` se expresa de la forma siguiente:

```
cin >> v;
```

2.8.2. Salida o escritura

La acción de **escritura** o **salida** de resultados comunica al exterior un valor, también por medio de un aparato (por ejemplo, el monitor del computador). La escritura de la expresión `E` se indica de la forma siguiente:

```
cout << E;
```

Existe una notación abreviada para escribir palabras y frases. Para escribir una concatenación de caracteres se encierra ésta entre comillas dobles. Por ejemplo, para mostrar en pantalla el mensaje

```
Introduzca la temperatura
```

no se realiza la secuencia de acciones

```
cout << 'I';
cout << 'n';
cout << 't';
cout << 'r';
cout << 'o';
cout << 'd';
...
```

sino la siguiente:

```
cout << "Introduzca la temperatura";
```

Pueden escribirse varias expresiones en la misma acción `cout`. Si se desea cambiar de línea se emplea la palabra `endl`. Por ejemplo:

```
cout << "X= " << x << endl <<  "Y=  " << y << endl;
```

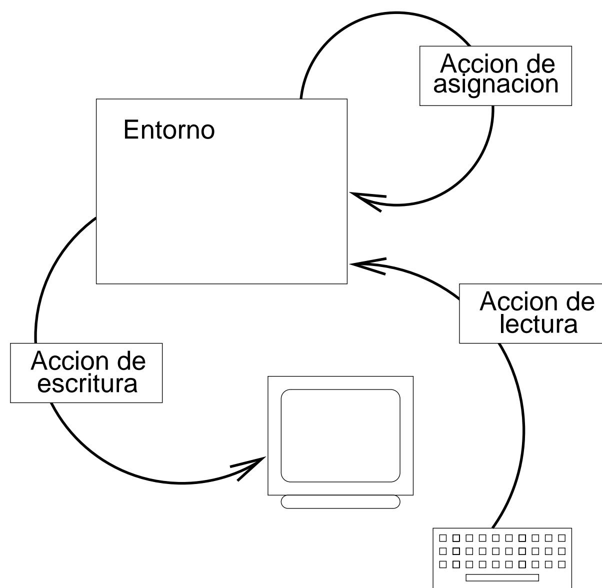

Figura 2.2: La acción de asignación es una función cuyos dominio y rango son el entorno del procesador. La acción de lectura modifica el entorno según la información producida por el teclado. La acción de salida no modifica el entorno y envía a la pantalla el valor de un indicador del entorno.

2.9. Composiciones de acciones

En el algoritmo anterior el trabajo ha consistido en una lista de acciones. Existen tareas más complejas en las que es necesario tomar decisiones y realizar acciones distintas en función de los valores de algunos indicadores. Existen tres composiciones de acciones: la composición **secuencial**, la composición **condicional** y la composición **iterativa**.

2.9.1. Composición secuencial. Bloques.

Para que el procesador ejecute varias acciones de forma consecutiva se escriben los nombres de las acciones uno a continuación del otro o bien en líneas separadas. Por ejemplo, el algoritmo que sigue hace que el procesador ejecute primero la acción `x=1;`. Después (tras terminar `x=1;`) la acción `x=x+1;`, y por último la acción `cout << x;`:

```
x=1; x=x+1; cout << x;
```

Llamaremos **composición secuencial** a una lista de acciones o composiciones de acciones, que se ejecutarán una tras otra. Definimos también **bloque** como una lista de definiciones de variables y constantes y de composiciones secuenciales *encerrada entre dos llaves*. Los siguientes son ejemplos de bloques:

```
{ x=x+1; y=y+cos(x); }
```

```
{ const int x=3; y=y+cos(x); }

{ short int x; x=3; short int y=x+1; }
```

Un programa sencillo en C++ consiste en un bloque, precedido de algunas definiciones cuyo sentido será explicado más adelante.

Ejemplo: el siguiente algoritmo sirve para convertir un temperatura, expresada en grados centígrados, a grados Fahrenheit:

```
#include <iostream>
using namespace std;
void main(void) {

  const float factor = 1.8;
  const float hielo_F = 32;

  float grados_C;
  float grados_F;

  cin >> grados_C;
  grados_F = grados_C*factor+hielo_F;
  cout << grados_F;
}
```

Este programa funciona como sigue: En primer lugar, el procesador solicita que el usuario introduzca por teclado el valor de una temperatura, que se almacena en la variable `grados_C`. A continuación se realiza el cálculo y se almacena el resultado en la variable `grados_F`. Por último, se muestra en pantalla el valor de `grados_F`.

La composición de acciones no es conmutativa. Observe que el primero de los algoritmos que siguen escribe el número 4 y el segundo escribe el número 3:

```
int V1;

V1 = 1;
V1 = V1+1;
V1 = V1*2;
cout << V1;
```

```
int V1;

V1 = 1;
V1 = V1*2;
V1 = V1+1;
cout << V1;
```

2.9.2. Composición condicional o alternativa

La composición alternativa permite elegir un procesamiento entre varios, en función del estado inicial. Hay tres tipos de composiciones condicionales: la alternativa simple, la alternativa doble y la alternativa múltiple. Las alternativas simple y doble se codifican mediante la orden `if`, la alternativa múltiple mediante la orden `switch`.

1. **Alternativa simple** Si una condición es cierta se ejecuta una acción o un bloque de acciones y si es falsa no se hace nada. La condición es una expresión de tipo booleano. Esta orden puede aparecer de las siguientes cuatro formas (la última se verá más adelante):

```
if (condición) acción
if (condición) bloque
if (condición) composición alternativa
if (condición) composición iterativa
```

Observe que no está permitido escribir una lista de acciones tras la condición, a menos que estén encerradas entre llaves.

En lo sucesivo, para indicar que en un punto de un programa puede aparecer una acción, o un bloque, o una composición alternativa o una composición iterativa emplearemos la siguiente notación:

```
if (condición) acción | bloque | alternativa | iterativa
```

Observe que en el ejemplo siguiente una de las acciones que se ejecuta es, a su vez, otra composición condicional. Diremos que estos condicionales están **anidados**.

```
#include <iostream>
using namespace std;
void main(void) {
    int x;
    cin >> x;
    if (x==2) x=x+1;
    if (x>3 && x<5) {
        cout << "x es mayor que 3";
        cout << "x es menor que 5";
    }
    if (x>3)
      if (x<5) {
        cout << "x es mayor que 3";
        cout << "x es menor que 5";
      }
}
```

2. **Alternativa doble:** Si una condición es cierta se ejecuta una acción o un bloque y si es falsa se ejecuta otra acción o bloque distinto. La condición también es una expresión booleana.

```
if (condición) acción | bloque | alternativa | iterativa
else acción | bloque | alternativa | iterativa
```

Ejemplo:

```
#include <iostream>
using namespace std;
void main(void) {
    int x;
    cin >> x;
```

```
    if (x>3) cout << "x es mayor que 3";
    else cout << "x es menor o igual que 3";

    if (x>8 || x<4) {
           cout << "x es mayor que 8";
           cout << "o x es menor que 4";
    } else
        if (x>6)
            cout << "x es 7 ó 8";

}
```

3. **Alternativa múltiple:** Similar a la alternativa doble, pero se elige entre varias alternativas en función de una expresión de tipo entero o carácter.

```
 switch (expresión) {
   case valor1:
            composición secuencial
            break;

   case valor2:
            composición secuencial
            break;

   ...

   default:
            composición secuencial
            break;
 }
```

Ejemplo:

```
#include <iostream>
using namespace std;
void main(void) {
  int x;
  cin >> x;
  switch (x) {
      case 1:  cout << "x es 1"; break;
      case 3:  cout << "x es 3"; break;
      case 8:  cout << "x es 8"; break;
      default: cout << "x no es 1" << endl;
               cout << "x no es 3" << endl;
               cout << "x no es 8" << endl;
               break;

  }
}
```

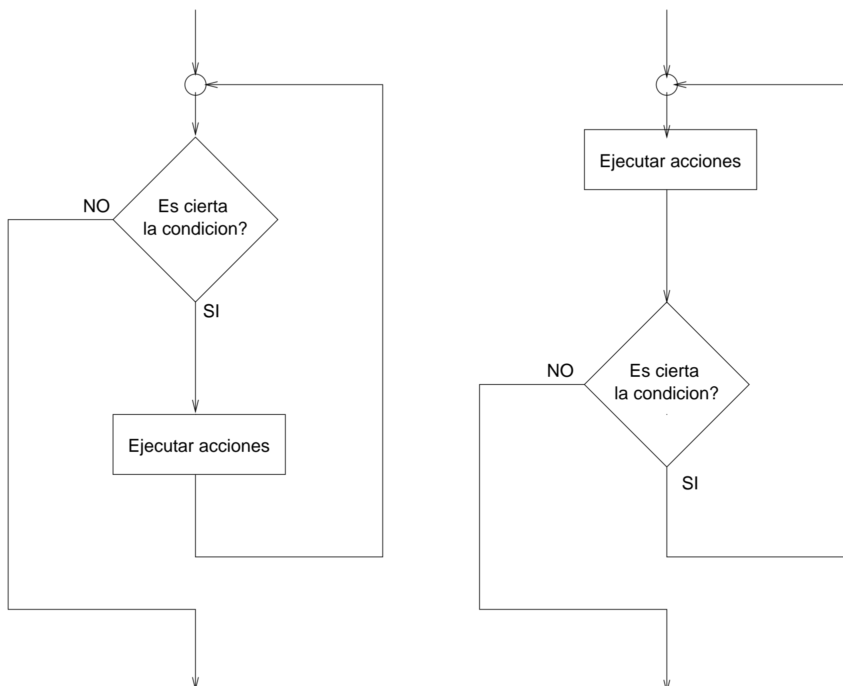

Figura 2.3: Diagramas de flujo de los bucles "while" (izquierda) y "do-while" (derecha)

2.9.3. Composición iterativa

La composición iterativa permite ejecutar una acción un número de veces que depende del estado inicial. Hay tres composiciones iterativas: la composición **while**, la composición **do-while** y la composición **for**.

1. **Composición while:** La composición `while` (ver figura 2.3, parte izquierda) repite una acción o una composición de acciones mientras una condición sea cierta. La notación es como sigue:

   ```
   while (condición) acción | bloque | alternativa | iterativa
   ```

   Ejemplo:

   ```
   #include <iostream>
   using namespace std;
   void main(void) {
      int x;
      x=1;
      while (x<1000) {
           x=x*2;
           cout << x;
      }
      cout << "final";
   ```

```
}
```

En este ejemplo, en primer lugar se evalúa `x<1000`, que es una expresión de tipo booleano. Si es falsa, se salta a la acción siguiente, que escribe la palabra "final". Si es cierta, se ejecuta `x=x*2;` y después `cout << x;`. Cuando `cout << x;` haya terminado, se vuelve a evaluar `x<1000` y se repite el proceso (es decir, si `x<1000` es `false`, se termina y si es `true` se vuelve a ejecutar el bloque, y así sucesivamente).

2. **Composición do-while:** La composición `do-while` (ver figura 2.3, parte derecha) ejecuta una acción o una composición de acciones, evalúa una condición y, si es cierta, vuelve a ejecutar las acciones. El proceso termina cuando la condición es falsa. La notación es como sigue:

```
do acción | bloque | alternativa | iterativa while (condición);
```

Ejemplo:

```
#include <iostream>
using namespace std;
void main(void) {
   int x;
   x=1;
   do {
        x=x*2;
        cout << x;
   } while (x<1000);
   cout << "final";
}
```

En este programa, en primer lugar se ejecuta el cuerpo del bucle (acciones `x=x*2;`, `cout << x;`). Cuando `cout << x;` termina, se evalúa `x<1000`, que es una expresión de tipo booleano. Si es falsa, se salta a la acción siguiente, `cout << "final";`. Si es cierta, se vuelven a ejecutar las dos acciones mencionadas, y así sucesivamente.

Observe que en la composición `do-while` el cuerpo del bucle se ejecuta al menos una vez, mientras que en la composición `while` puede ser que no se ejecute ninguna acción.

3. **Composición for:** La composición `for` (ver figura 2.4) es una forma alternativa de escribir la composición `while` que, en general, se emplea cuando se desea que se repita una acción o una composición de acciones cierto número de veces. La notación es como sigue:

```
for (A;B;C) D
```

y es equivalente a

```
A; while (B) { D C; }
```

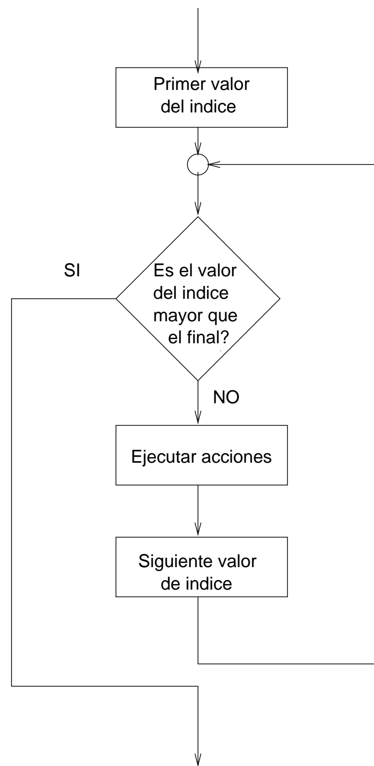

Figura 2.4: Diagrama de flujo del bucle “para”

Por ejemplo, los dos programas que siguen son equivalentes:

```
for (i=0;i<=10;i++)          i=0;
   cout << i << endl;        while (i<=10) {
                               cout << i << endl;
                               i++;
                             }
```

Es frecuente que el bucle `for` tenga la siguiente forma:

```
for (variable=valor inicial;
     variable<=valor final;
     variable=variable+incremento)
   acción | bloque | alternativa | iterativa
```

Por ejemplo:

```
#include <iostream>
using namespace std;
void main(void) {
    int i;
    for (i=1;i<=10;i++) cout << i;
    cout << "final";
}
```

en primer lugar se realiza la asignación `i=1`. A continuación se comparan los valores de `i` y `10`. Si `i`≤`10`, se ejecuta la acción `cout << i;` . A continuación se le suma una unidad a la variable `i` y se vuelve al punto en que se compararon `i` y `10`. Cuando `i`>`10`, se salta a la acción `cout << "final";` .

2.10. Comentarios

Un comentario es un fragmento de código que el compilador ignora. Los comentarios en C++ van precedidos de // si ocupan una línea y van encerrados en un par /* */ si ocupan varias.

Ejemplo:

```
#include <iostream>
using namespace std;
void main(void) {
      a = 1;      // Asignacion de 1 a la variable a
      /* este comentario ocupa
         dos lineas */
      b = 2;
      cout << a << " " << b << endl;
}
```

2.11. El tipo vector

El entero, el real, el carácter o el booleano son tipos simples. Por el contrario, el complejo es un tipo estructurado, porque se define mediante un par de valores que, a su vez, pertenecen a otro tipo (el real). En otras palabras, el tipo complejo es el producto cartesiano del tipo real por sí mismo o, en notación matemática, $\mathbf{C} = \mathbf{R} \times \mathbf{R} = \mathbf{R}^2$. Decimos que $\mathbf{R}$ es el **tipo base** del complejo.

El tipo **vector** o **array** (en este curso emplearemos ambos nombres indistintamente) es el producto cartesiano de un tipo base por sí mismo n veces. Si el tipo base del vector es el conjunto $\mathbf{T}$, el tipo "vector de n elementos de $\mathbf{T}$" es $\mathbf{T}^n$, o lo que es lo mismo, cada elemento del tipo vector es una tupla de n elementos del tipo base (por ejemplo, un complejo es un vector de reales de dimensión 2).

El tipo base de un vector puede ser cualquier otro, ya sea simple o estructurado. Podemos utilizar vectores de reales, de caracteres (que se emplean para almacenar textos), de booleanos (que se emplean para representar conjuntos) o de vectores. Un vector de vectores es una matriz.

Para declarar una variable llamada `v` del tipo "vector de n elementos de $\mathbf{T}$" (es decir, $v \in \mathbf{T}^n$) se emplea la siguiente notación:

```
vector<T> v(n);
```

y se escribe la directiva `#include <vector>` al comienzo del programa. Por ejemplo, un vector de $\mathbf{R}^3$ con nombre `x` se declara así:

```
#include <vector>
int main() {
        vector<float> x(3);
        etc.
}
```

En álgebra se emplea un subíndice para referirse a las componentes de un vector. Por convenio, numeraremos las componentes empezando en el cero y no en el uno. Si $v \in \mathbf{R}^4$, entonces $v = (v_0, v_1, v_2, v_3)$. En C++ se indican las componentes de un vector mediante su número encerrado entre corchetes; Por ejemplo, para hacer referencia a v_2 se emplea el nombre `v[2]`. Si $v \in \mathbf{R}^4$ entonces $v_2 \in \mathbf{R}$, y de igual forma si `v` es un vector de 4 reales entonces `v[0]`, `v[1]`, `v[2]` y `v[3]` son variables de tipo real. Por ejemplo,

```
vector<float> v(3); // vector de tres reales: v[0], v[1] y v[2]
vector<char> a(32); // vector de 32 caracteres, entre a[0] y a[31]
vector<float> z();  // vector de 0 reales (sin componentes)
vector<float> z(0); // ERROR
```

Si se desea dar un valor inicial a los elementos de un vector se emplea la siguiente notación:

```
vector<float> v(3,0.0); // v[0]=0.0; v[1]=0.0; v[2]=0.0;
```

La dimensión de un vector se puede consultar mediante la orden `size`, de la forma siguiente:

```
int i;
vector<float> v(3,0.0);
for (i=0; i<v.size(); i++) {  // igual que for (i=0;i<3;i++) {
   cout << v[i] << endl;
}
```

Como se ha dicho, las matrices se representan por vectores de vectores. Por ejemplo,

```
// Vectores
vector<float> fila(4); // 4 componentes
vector<float> fila1(5,2.3); // 5 componentes, valor inicial 2.3

// Matrices
vector<vector<float> > m1(3); // 3 filas, 0 columnas
vector<vector<float> > m2(3,4); // ERROR
vector<vector<float> > m3(3,fila); // 3 filas, 4 columnas
vector<vector<float> > m3(3,vector<float>(4)); // Lo mismo
vector<vector<float> > m4(3,fila1); // 3 f. 5 c., valor inicial 2.3
vector<vector<float> > m4(3,vector<float>(5,2.3)); // Lo mismo
```

2.11.1. Recorrido de los elementos de un vector

Es frecuente que se emplee la composición `for` si es necesario ejecutar acciones que afecten a todos los elementos de un vector. Por ejemplo, para pedir por teclado el valor de un vector de 10 elementos se lee por separado cada una de sus componentes. Para esta tarea y otras similares es más sencillo utilizar un bucle `for` que un bucle `while`:

```
int i;
vector<float> v(10);
for (i=0;i<v.size();i++) cin >> v[i];

int i;
vector<float> v(10);
i=0;
while (i<v.size()) {
    cin >> v[i];
    i=i+1;
}
```

Se dice que un algoritmo realiza el **recorrido** de un vector cuando opera con todos los elementos del vector, uno tras otro y en orden. El siguiente programa recorre un vector:

Ejemplo 1: Pedir por teclado un vector de 10 componentes y calcular su módulo.

```
#include <iostream>
#include <vector>
using namespace std;
void main(void) {
    int i;
    vector<float> v(10);
    float suma_cuad=0;
    for (i=0;i<v.size();i++) cin >> v[i];
    for (i=0;i<v.size();i++) suma_cuad=suma_cuad+v[i];
    cout << "El módulo es " << sqrt(suma) << endl;
}
```

Cada elemento de un vector de vectores es un vector y puede aplicársele la orden `size`, como puede verse en el ejemplo que viene a continuación.

Ejemplo 2: Pedir por teclado y sumar las componentes de una matriz de 5×5 enteros.

```
#include <iostream>
#include <vector>
using namespace std;
void main(void) {
   vector<vector<int> > a(5,vector<int>(5)),i,j;
   for (i=0;i<a.size();i++)
     for (j=0;j<a[i].size();j++) {
        cout << "Dame el elemento ("
             << i << "," << j << ")" << endl;
        cin >> a[i][j];
     }
   float suma=0;
   for (i=0;i<a.size();i++)
     for (j=0;j<a[i].size();j++) {
           suma=suma+a[i][j];
     }
   cout << "La suma es " << suma << endl;
}
```

3

Definición de Acciones no Primitivas

3.1. Estructura de un algoritmo

Diseñar un algoritmo consiste en seleccionar varias acciones elementales y en organizarlas en el tiempo para obtener el resultado correspondiente a las acciones acumuladas. Un algoritmo es también una acción. Como sólo hay tres formas básicas de organizar acciones (secuencial, condicional e iterativa) un algoritmo es:

- una acción primitiva, o
- una composición secuencial de acciones, o
- una composición condicional de acciones, o
- una composición iterativa de acciones

donde, a su vez, las acciones que forman parte de las composiciones pueden descomponerse según el mismo esquema (ver figura 3.1). Esta sintaxis produce organizaciones en bloques anidados.

Ejemplo: El algoritmo que sigue escribe los factoriales de los diez primeros naturales:

```
#include <iostream>
using namespace std;
void main(void) {
   int i,j,f;
   for (i=1;i<=10;i++) {
     f=1;
     for (j=1;j<=i;j++) f=f*j;
     cout << "El factorial de " << i << " es " << f << endl;
```

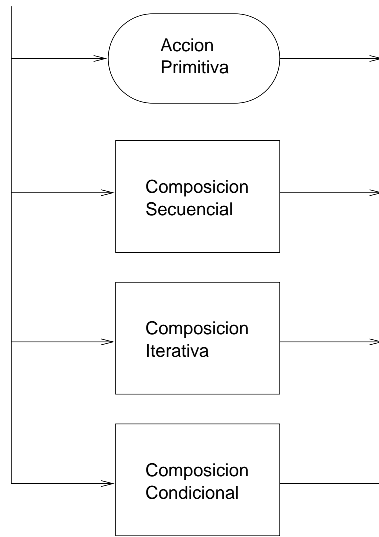

Figura 3.1: Diagrama sintáctico de una accion. Una acción puede ser primitiva o bien ser una composición secuencial, condicional o iterativa de otras acciones, que a su vez pueden ser primitivas o nuevas composiciones.

```
   }
}
```

La estructura de este algoritmo está en la figura 3.2. Obsérvese que está organizado en bloques anidados, cada uno de los cuales es una acción. Las acciones no primitivas son composiciones de otras acciones y las relaciones de inclusión que unos bloques tienen con otros forman una estructura de árbol como la mostrada en la figura 3.3.

3.2. Ambito de una declaración

Es posible declarar variables y constantes en cualquier punto de un programa C++. Una variable no puede ser empleada antes de ser declarada. Hasta ahora hemos definido las variables al comienzo del programa:

```
#include <iostream>
using namespace std;
void main(void) {
  int i,j;
  i=3;
  j=1;
  cout << i << " " << j << endl;
}
```

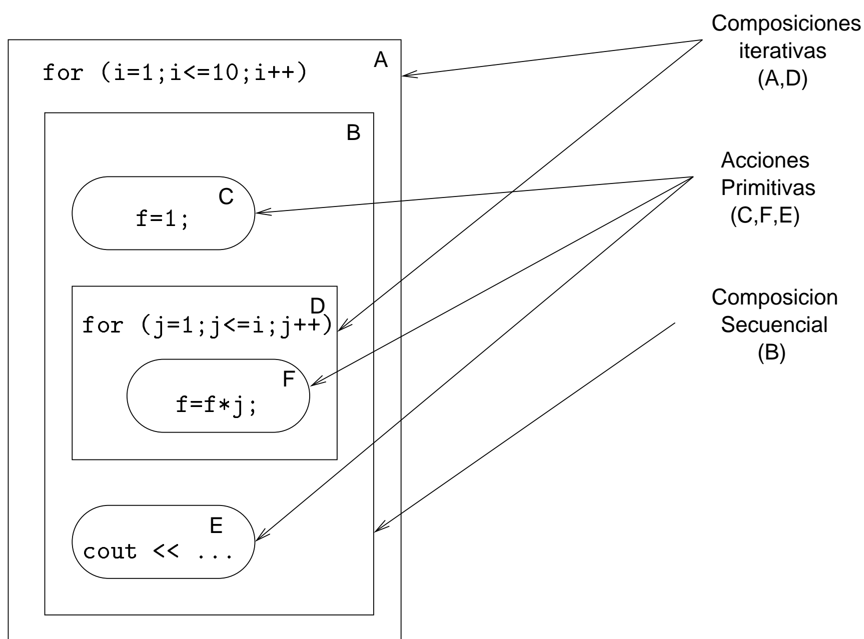

Figura 3.2: Diagrama sintáctico del algoritmo de la sección 3.1. Las acciones primitivas tienen los bordes redondeados. Cada acción está nombrada por una letra. C, F y E son acciones primitivas. A y D son composiciones iterativas. B es una composición secuencial.

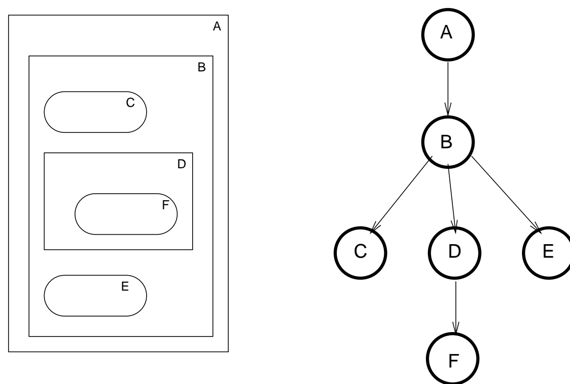

Figura 3.3: Arbol de relaciones de inclusión entre los bloques que componen un algoritmo. Las flechas significan "contiene a". La estructura es un árbol porque no hay dos bloques que contengan simultáneamente a otro.

pero también se pueden declarar justo antes de ser utilizadas por primera vez:

```
#include <iostream>
using namespace std;
void main(void) {
  int i;
  i=3;
  int j;
  j=1;
  cout << i << " " << j << endl;
}
```

El **ámbito** de una variable (o, en general, de un identificador) es la parte del programa en que la variable (o el identificador) puede usarse. Por ejemplo, en el programa anterior el ámbito de la variable `j` comienza en la quinta línea y llega hasta el final del programa.

Si un identificador se declara dentro de un bloque, su ámbito no llega hasta el final del programa, sino hasta el final del bloque. Si hay otros bloques definidos dentro de este primero, la variable también es visible desde ellos. Por ejemplo, el siguiente programa es incorrecto, porque el ámbito de la variable `j` termina en la línea octava (donde se cierra la llave del `if`):

```
#include <iostream>
using namespace std;
void main(void) {
  int i;
  i=3;
  if (i>2) {
     int j;
     j=1;
  }
  cout << i << " " << j << endl;
}
```

El programa que sigue escribe en pantalla los números 3, 1 y 5.

```
#include <iostream>
using namespace std;
void main(void) {
  int i,j;
  i=1; j=5;
  if (i<2) {
     int j;
     j=3;
     cout << j << endl;
  }
  cout << i << " " << j << endl;
}
```

En la línea 8 existen dos variables llamadas 'j', una de ellas con valor 5 y otra con valor 3. Por convenio se elige la declaración más reciente y por tanto la orden `cout`

`<< j << endl;` escribe un 3. La orden `cout << i << " "  << j << endl;` no está en el ámbito de la variable declarada en la línea 6, y por tanto en este punto solamente existe una variable con nombre `j`.

Tampoco es necesario que las declaraciones de las variables estén dentro del bloque `main`. El siguiente programa también es correcto:

```
#include <iostream>
using namespace std;
int i,j;
void main(void) {
  i=1; j=5;
  if (i<2) {
     int j;
     j=3;
     cout << j << endl;
  }
  cout << i << " " << j << endl;
}
```

Esta última forma de declarar variables sólo tiene sentido cuando hay más de un bloque y se desea que las variables `i` y `j` sean visibles desde todos ellos. Se dice que las variables `i` y `j` son **globales**. Su utilidad se verá en la sección 3.4 de este capítulo.

En C++ las variables globales existen durante toda la ejecución del programa, mientras que las no globales (a las que llamaremos variables **locales**) se crean en el punto del programa en que se definen, y se destruyen en cuanto dejan de ser visibles (es decir, tras ejecutarse la última instrucción del bloque en el que se hayan definido).

3.3. Diseño descendente

La técnica de diseño conocida como **diseño descendente** se basa en construir el algoritmo comenzando por la parte superior del árbol de la figura 3.3 y **refinar** sucesivamente las especificaciones de las acciones intermedias hasta que se correspondan con acciones primitivas del procesador. Las acciones intermedias no pertenecen al léxico del procesador.

El primer paso en el diseño descendente del algoritmo de la sección 3.1 podría ser:

```
#include <iostream>
using namespace std;
void main(void) {
   int i;
   for (i=1;i<=10;i++) {
      escribir factorial de i
   }
}
```

donde “escribir factorial de i” es una acción intermedia que no pertenece al lenguaje C++. El siguiente refinamiento consiste en hacer un algoritmo que resuelva únicamente el problema “escribir factorial de i”:

```
int f,j;
f=1;
for (j=1;j<=i;j++) f=f*i;
cout << "El factorial de " << i << " es "  << f << endl;
```

y, por último, introducir esta acción en su sitio:

```
#include <iostream>
using namespace std;
void main(void) {
   int i;
   for (i=1;i<=10;i++) {
      // escribir factorial de i
      int f,j;
      f=1;
      for (j=1;j<=i;j++) f=f*i;
      cout << "El factorial de " << i << " es "  << f << endl;
   }
}
```

3.4. Acciones con nombre

En todas las etapas intermedias del diseño descendente el algoritmo contiene acciones no primitivas. Las acciones no primitivas se reemplazan por composiciones de acciones primitivas o por composiciones de otras acciones no primitivas más sencillas.

Si en el diseño aparece la misma acción no primitiva en varias partes distintas, es razonable pensar en darle un nombre a esta acción e incorporarla al léxico del procesador.

Ejemplo: Diseñar un algoritmo que imprima los factoriales de los números del 10 al 20 y del 100 al 200.

El bloque más externo es una composición secuencial:

```
void main(void) {
   int n;
   escribir n! para n=1...10
   escribir n! para n=20...25
}
```

Cada una de las acciones de la composición secuencial es un bucle `for`:

```
void main(void) {
    int n;
    for (n=1;n<=10;n++) {
       escribir n!
```

```
    }
    for (n=20;n<=25;n++) {
       escribir n!
    }
}
```

y la versión final es esta:

```
void main(void) {
    int n;
    for (n=1;n<=10;n++) {
       int j,f=1;
       for (j=1;j<=n;j++) f=f*j;
       cout << f << endl;
    }
    for (n=20;n<=25;n++) {
       int j,f=1;
       for (j=1;j<=n;j++) f=f*j;
       cout << f << endl;
    }
}
```

en la que hay una parte repetida. Por esta razón, definimos la palabra `escribir_fact`, equivalente al algoritmo

```
       int j,f=1;
       for (j=1;j<=n;j++) f=f*j;
       cout << f << endl;
```

En C++ se puede hacer esto mediante la siguiente construcción:

```
void nombre_de_la_palabra(lista de parámetros) {
   algoritmo
}
```

El programa queda como se muestra a continuación:

```
#include <iostream>
using namespace std;
void escribir_fact_n(int n) {
       int j,f=1;
       for (j=1;j<=n;j++) f=f*j;
       cout << f << endl;
}
void main(void) {
    for (n=10;n<=20;n++) {
      escribir_fact(n);
    }
    for (n=100;n<=200;n++) {
      escribir_fact(n);
    }
}
```

Las variables que hayan sido introducidas para **parametrizar** la acción no se declaran como las demás, sino en la definición de la acción, a continuación del nombre de ésta y entre los paréntesis. La variable `n` de la acción `escribir_factorial` es un **parámetro formal**. Un parámetro formal es una variable que se crea en el instante en que se empieza a ejecutar la acción y que se destruye al terminar ésta.

Observa el siguiente ejemplo:

```
#include <iostream>
using namespace std;
void escribir_factorial(int x) {
   int f=1,j;
   for (j=1;j<=x;j++) {
         f=f*j;
   }
   cout << f << endl;
}

void main(void) {
   int n;
   for (n=1;n<=5;n++) {
      escribir_factorial(2*n);
   }
   for (n=1;n<=5;n++) {
      escribir_factorial(2*n-1);
   }
}
```

Cada valor inicial del parámetro formal (`2*n` y `2*n-1` en este ejemplo) se denomina **parámetro real**. Observa que el parámetro real puede ser una cualquier expresión del mismo tipo que el parámetro formal.

Una acción puede tener más de un parámetro. En ese caso, las definiciones de los parámetros se separan mediante comas (,). Por ejemplo:

```
#include <iostream>
using namespace std;

void escribecomb(int n, int m) {
   int i,num=1,den=1;
   for (i=n-m+1;i<=n;i++) num=num*i;
   for (i=2;i<=m;i++) den=den*i;
   cout << num/den;
}
void main(void) {
   int N,M;
   for (N=2;N<=10;N++) {
     for (M=0;M<=N;M++) {
         escribecomb(N,M);
         cout << " ";
     }
```

```
        cout << endl;
    }
}
```

3.5. Funciones y operadores

En la lección 2 se estudiaron los conceptos de expresión aritmética y lógica, y con ellos se introdujeron las nociones de función (como sin(), cos(), etc.) y operador (como $+$, $-$, etc.). No hay diferencia conceptual entre función y operador. Un operador es una notación más cómoda para una función con dos argumentos. Por ejemplo, es más claro escribir $2+3$ (operador) que $+(2,3)$ o suma$(2,3)$ (funciones). En C++ es posible definir nuevas funciones y nuevos operadores.

3.5.1. Declaración y definición de funciones

En la declaración de una función se da el nombre de la función, el tipo del valor que devuelve y el número y los tipos de los argumentos que deben ser suministrados en la llamada a la función. En la definición se da también el algoritmo que debe ejecutarse para calcular la imagen de los argumentos de la función.

Cada función que sea llamada en un programa debe ser definida en algún punto de este y sólo en uno (aunque puede estar declarada en más de un punto) y no puede emplearse antes de estar declarada o definida.

Por ejemplo,

```
int f(int, int);
```

es la *declaración* de una función `f: int × int → int`. No se indica qué instrucciones deben ejecutarse para calcular $f(a,b)$ dados dos enteros a y b. Tan solo se declara que f tiene dos argumentos de tipo entero y que produce otro número entero.

La declaración de una función puede contener nombres de argumentos, pero el compilador los ignora:

```
int f(int a, int b);
```

Las declaraciones de funciones también se llaman **prototipos**.

La definición de la función es un bloque. El argumento de salida se define mediante mediante la orden `return`. Por ejemplo, la función $f(x,y) = 2x + y$ se codifica en C++ así:

```
int f(int x, int y) {
    int resultado;
    resultado=2*x+y;
    return resultado;
}
```

o bien así:

```
int f(int x, int y) {
   return 2*x+y;
}
```

En el siguiente programa se declara la función f (su prototipo), a continuación se utiliza y por último se define:

```
#include <iostream>
using namespace std;
int f(int, int);

void main(void) {
   int a=f(2,3);
   cout << a << endl;
}

int f(int x, int y) {
   return 2*x+y;
}
```

y este programa también pudo escribirse eliminando la declaración: en primer lugar se define y después se utiliza:

```
#include <iostream>
using namespace std;
int f(int x, int y) {
   return 2*x+y;
}
void main(void) {
   int a=f(2,3);
   cout << a << endl;
}
```

Observe que la definición de un procedimiento es la misma que la de una función que devuelve el tipo `void`.

3.5.2. Operadores

Un operador se define de la misma forma que una función, pero su símbolo va precedido de la palabra `operator`. Suponga que `T` es el nombre de un tipo. Entonces las siguientes son definiciones de operadores:

```
T operator+(T a, T b);
T operator[](T x,int y);

float f(T y,T z, T u, int w) {
        T x;
        x=y+z;          // x=operator+(y,z);
        x=x+u[w];       // x=operator+(x,operator[](u,w));
        return x;
}
```

Pueden usarse los siguientes símbolos (el significado de alguno de ellos no se verá hasta más adelante):

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>^</td><td>x</td><td>&</td></tr>
<tr><td>|</td><td>~</td><td>!</td><td>=</td><td><</td><td>></td><td>+=</td></tr>
<tr><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td>^=</td><td>&=</td><td>|=</td></tr>
<tr><td>>=</td><td>&&</td><td>||</td><td>++</td><td>--</td><td>->*</td><td>,</td></tr>
<tr><td>-></td><td>[]</td><td>()</td><td>new</td><td>new[]</td><td>delete</td><td>delete[]</td></tr>
</table>

En el lenguaje C++ **al menos uno de los argumentos de un operador debe ser un tipo definido por el usuario** (este concepto se estudiará en la lección 4) porque el significado de todos los operadores de la tabla anterior ya está establecido para los tipos predefinidos y cambiarlo puede llevar a ambigüedades.

3.5.3. Sobrecarga

El uso el mismo nombre para una función, procedimiento u operador que realiza diferentes tareas sobre diferentes tipos de argumento se denomina **sobrecarga**. Por ejemplo, las divisiones de enteros y de reales comparten el mismo símbolo. El compilador sabe cuál de ellas debe utilizar por el tipo de los argumentos. En C++ las acciones, las funciones y los operadores definidos por el usuario también pueden estar sobrecargados.

Dos funciones/acciones/operadores con el mismo nombre/símbolo deben distinguirse en el número y/o en el tipo de sus argumentos. No es válido que se diferencien solamente en el tipo del argumento de salida.

3.6. Sinónimos, referencias y punteros

En la sección 2.1 se indicó que toda variable tiene al menos un nombre y sólo un valor. Las acciones acceden a los valores de los objetos a través de su nombre. Por ejemplo, suponga que la variable llamada `x` es de tipo entero y se corresponde con la dirección de memoria 999. Si se desea almacenar el valor 12 en esa variable, o lo que es lo mismo, ejecutar la acción

```
x=12;
```

se codifica el valor 12 de acuerdo con el tipo de la variable,

$$12 = 00000000000000000000000000001100$$

y se almacena el patrón de bits correspondiente en la dirección de memoria 999.

Cuando se convierte un programa C++ al lenguaje de máquina, el compilador asigna una dirección a cada variable. Para ello crea una tabla en la que cada uno de los nombres (como `x`) se relaciona con un número, el de la posición de memoria en que se almacena su contenido (ver figura 3.4).

Es posible hacer que el compilador asocie la misma posición de memoria a dos nombres de variable distintos. En este caso, el objeto de almacenamiento que está en esa posición de memoria tiene dos nombres. También se puede decir que los dos nombres son **sinónimos** o que el último nombre de la tabla es una **referencia** al primero. En C++ esto se programaría de la siguiente forma:

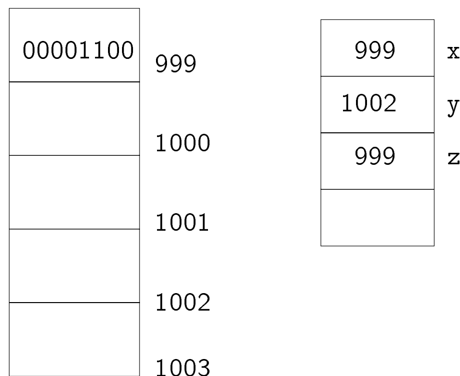

Figura 3.4: Tabla en que se relaciona el nombre de cada variable con su contenido. El contenido de la variable x se almacena en la posición de memoria 999. La variable y se almacena en la posición 1002. La variable z se almacena en la posición 999, la misma que x, luego se puede decir que z y x son dos nombres del mismo objeto de almacenamiento, que z y x son sinónimos o que z es una referencia a x. Esta tabla solamente existe mientras se está compilando el programa C++.

```
#include <iostream>
using namespace std;
void main(void) {
  int x;
  x=12;
  int &z=x;
  z=20;
  cout << x << endl;
}
```

En la línea `int &z=x;` se está indicando que z es una referencia a x, es decir, que tanto x como z son dos nombres del mismo objeto de almacenamiento. La dirección de este objeto es la que se asignó a x.

Por otra parte, el conjunto de todas las direcciones de memoria en que puede estar un objeto de un tipo T es un tipo de datos a su vez (un subconjunto de los números enteros). Este tipo recibe el nombre de **puntero a T**. En C++ definimos una variable de tipo "puntero a T" de la forma siguiente:

```
T *nombre_variable;
```

Por ejemplo,

```
float *px;
```

es una declaración de una variable llamada px de tipo "puntero a real". Si 999 es el número de una posición de memoria en la que puede almacenarse una variable, la orden

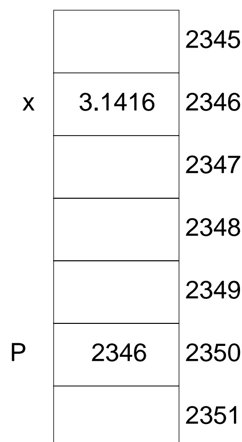

Figura 3.5: La variable real `x` está en almacenada en la dirección 2346 y contiene 3.1416. La variable `P`, de tipo `puntero a real`, está almacenada en la dirección 2350 y contiene 2346, que es la dirección en que está `x`. Por tanto, `*P` es un sinónimo de `x`.

```
px=999;
```

hace que el valor de `px` sea 999. Una operación como ésta no es habitual en los programas que haremos en este curso. Es más corriente hacer

```
float x=4.5;
float *px=&x;
```

donde el operador "&" significa "la dirección de". Es decir, el valor de la variable `px` es el número de la posición de memoria en que se guarda la variable `x`. El operador "*" es, en cierto sentido, el inverso de "&". La expresión "`*px`" es un nuevo nombre para el objeto de almacenamiento contenido en la dirección que está guardada en `px`. Por ejemplo, si se hace

```
#include <iostream>
using namespace std;
void main(void) {
  float x=4.5;
  float *px=&x;
  *px=2.2;
  cout << x << endl;
}
```

se muestra en pantalla el número 2.2. En este caso, `*px` y `x` son sinónimos temporalmente, hasta que `px` cambie de valor (ver figura 3.5).

3.7. Tiempo de vida de una variable

Las variables definidas dentro de una acción, función u operador son locales, y por tanto se crean justo antes de que esa acción comience a ejecutarse y se

destruyen tras terminar esta. Es posible cambiar este comportamiento declarando una o más variables locales a la acción con el atributo **static**. Una variable estática es una variable global con ámbito restringido: no se destruye y conserva su valor entre dos llamadas consecutivas a la acción. Por ejemplo, el programa siguiente escribe en pantalla $\sum_{j=1}^{i} j$ para valores de i entre 1 y 10:

```
#include <iostream>
using namespace std;

void escribesuma(int x) {
    static int y;
    y=y+x;
    cout << y << endl;
}

void main(void) {
   int i;
   for (i=1;i<10;i++) escribesuma(i);
}
```

A diferencia de las demás variables (a las que llamaremos **automáticas**) una variable estática tiene inicialmente el valor cero, salvo que indiquemos lo contrario.

3.8. Mecanismos del paso de argumentos

Al explicar la parametrización de acciones se dijo que pasar un argumento era equivalente a definir una variable nueva, asignarle el valor del parámetro real y utilizar esa variable dentro de la acción. Si se hacía algún cambio a esa variable nueva, el parámetro real no cambiaba.

Sin embargo, si se pasa un puntero a una variable como argumento a una acción, es posible que la acción modifique el contenido de una variable definida en un bloque externo. Por ejemplo:

```
#include <iostream>
using namespace std;
void intercambia(int *a, int *b) {
  int c;
  c=*a; *a=*b; *b=c;
}

void main(void) {
  int x=1,y=2;
  intercambia(&x,&y);
  cout << x << " " << y << endl;
}
```

La acción `intercambia` tiene como argumentos las direcciones de dos variables de tipo entero. Por esta razón, desde el bloque `main` se la invoca con la línea `intercambia(&x,&y);` (recuerde que el operador `&` produce la dirección en que

se encuentra una variable). Por tanto, la línea `c=*a; *a=*b; *b=c;` intercambia los contenidos de las variables *x* e *y* y el algoritmo escribe los números 2 y 1.

Esto mismo puede escribirse de forma abreviada, con la notación que se indica en el ejemplo que sigue:

```
#include <iostream>
using namespace std;

void intercambia(int &a, int &b) {
  int c;
  c=a; a=b; b=c;
}

void main(void) {
  int x=1, y=2;
  intercambia(x,y);
  cout << x << " " << y << endl;
}
```

Distinguiremos entonces dos *mecanismos de paso de argumentos*:

- Si no se utiliza el símbolo `&`, se crea una nueva variable por cada parámetro formal y se copia en ella el valor del parámetro real.
- Si se utiliza el símbolo `&`, se hace que cada parámetro formal sea una referencia a su parámetro real correspondiente.

El primero de los mecanismos se conoce como **paso por valor** y el segundo como **paso por referencia**.

En resumen, los argumentos pueden pasarse por dos mecanismos: por valor y por referencia. Si una acción modifica el valor de un argumento que se ha pasado por valor, a la salida de la acción este argumento conserva su valor inicial. Si el argumento se ha pasado por referencia, el valor del parámetro real es el que se le ha dado dentro de la acción.

3.9. Semántica del paso de argumentos

El mecanismo del paso de argumentos define la forma en que el procesador realiza el paso. La semántica del paso de argumentos se refiere a la intención del programador acerca de si el argumento es un dato necesario para el cálculo, una variable en la que devolver un resultado o ambas cosas a la vez. Por ejemplo, el argumento `x` de la acción siguiente es un dato:

```
void escribir_factorial(int x) {
   int f=1,j;
   for (j=1;j<=x;j++) {
         f=f*j;
   }
   cout << f << endl;
}
```

En esta otra acción, `x` es un dato, pero `y` es un resultado:

```
#include <iostream>
using namespace std;
void calcular_factorial(int x,int &f) {
   f=1,j;
   for (j=1;j<=x;j++) {
         f=f*j;
   }
}
void main(void) {
   int x=3,y;
   calcular_factorial(x,y);
   cout << "El factorial de " << x << " es " << y << endl;
}
```

y, por último, en esta son tanto datos como resultados:

```
#include <iostream>
using namespace std;
void intercambia(int &a, int &b) {
  int c;
  c=a; a=b; b=c;
}

void main(void) {
  int x=1, y=2;
  intercambia(x,y);
  cout << x << " " << y << endl;
}
```

Los parámetros que sólo se usan como datos se llaman **datos de entrada**. Los que sólo se emplean para depositar en ellos un resultado se denominan **datos de salida**. Los que son un dato y además se modifican se denominan **datos de entrada y salida**.

Puede hacerse que el compilador C++ dé un error si se intenta modificar el valor de un argumento de entrada. Para esto se antepone la palabra `const` al nombre del parámetro. Por ejemplo:

```
void calcular_factorial(const int x,int &f) {
   f=1,j;
   for (j=1;j<=x;j++) {
         f=f*j;
   }
}
```

Las reglas que se aplican para decidir entre los mecanismos de paso por valor y por referencia son las siguientes:

- Los argumentos de entrada se pueden pasar de dos formas:

  - Por valor.
  - Por referencia, pero precedidos siempre de la palabra `const` (por ejemplo: "`const int &x`").

- Los argumentos de entrada/salida y de salida se pasan por referencia.

3.10. Argumentos procedurales

Las acciones, las funciones y los operadores son algoritmos que, una vez convertidos al lenguaje del microprocesador, están almacenados en la memoria junto con los datos. Por esta razón tiene sentido hablar del tipo "puntero a una función" o "puntero a una acción" como el conjunto de los números de las celdas de memoria en que empiezan las codificaciones de las funciones y acciones definidas en un programa.

En C++ el nombre de una función se evalúa como el número de la celdilla en que comienza su código. Si se define una función `int f(float x)`, entonces `f(3.2)` es un número entero, la imagen de 3.2, mientras que `f` es un número distinto, la posición de memoria en que empieza el código de la función `f`.

La sintaxis de la definición de variables de tipo "puntero a función" se muestra en el siguiente ejemplo:

```
#include <iostream>
using namespace std;
int f(float y) {
    return int(2*y);
}
void main(void) {
    int (*pf)(float);       // Puntero a función de R en Z
    int z;
    pf=f;                   // pf1 guarda la dirección de x
    z=(*pf)(2.3);           // Llamada a la función x
    cout << z << endl;
}
```

Los punteros a función sirven para que una acción sea parámetro de otra acción, aunque esto no es muy frecuente. En el ejemplo siguiente se calcula la derivada numérica de una función, de forma aproximada:

```
#include <iostream>
#include <cmath>
using namespace std;

float f1(float x) { return 2*x; }
float f2(float x) { return cos(x); }

const float delta=0.001;
float derivada(float (*pf)(float),float x) {
```

```
  return ((*pf)(x+delta)-(*pf)(x))/delta;
}

void main(void) {

   cout << derivada(f1,0.3) << endl;
   cout << derivada(f2,0.3) << endl;

}
```

3.10.1. Modularidad y ocultación de información

En programas muy extensos o escritos por varias personas distintas es útil disponer de variables, constantes y acciones que sean visibles en un subconjunto de las acciones que se hayan definido e invisibles en el resto. Para ello se divide un programa en **módulos**. El ámbito de una variable, constante o acción definida en un módulo es el cuerpo del módulo.

La construcción C++ para el módulo la orden `namespace`. Por ejemplo:

```
namespace nombre {
    double f1(bool a) { /* .... */ }
    double f2(int a,float b) { /* ... */ }
    int f3(int a, int b) { /* ... */ }
}
```

Si la definición de las funciones es larga es más legible dejar dentro del namespace las declaraciones y sacar las definiciones fuera. Para indicar que se hace referencia a la función `f` definida en el módulo `m` se emplea la notación "`m::f`". Por ejemplo:

```
namespace nombre {
    // Sólo declaraciones
    double f1(bool a);
    double f2(int a,float b);
    int f3(int a, int b);
}
// Definiciones fuera del bloque
double nombre::f1(bool a) { /* ... */ };
double nombre::f2(int a,float b) { /* ... */ };
int nombre::f3(int a, int b) { /* ... */ };
```

La orden `using` introduce un sinónimo local de uno o todos los nombres del namespace:

```
   using nombre::f1;              // f1 es sinónimo de nombre::f1
   double x=f1(true)

   using namespace nombre;        // sinónimos locales de f1, f2, f3
```

3.11. Recursividad

La recursividad es una característica de los lenguajes de programación en la cual se permite que un procedimiento o función haga uso de sí mismo dentro de su definición.

La recursividad, junto con la iteración, son los dos mecanismos que se suministran para describir cálculos que, con pequeñas variaciones, han de repetirse un cierto número de veces. La recursividad es la forma más sencilla de programar una definición inductiva de una tarea, aunque generalmente también es la más ineficiente. Por ejemplo, a partir de la definición inductiva de factorial

$$x! = \begin{cases} \text{si } x > 0 & \text{x(x-1)!} \\ \text{si } x = 0 & 1 \end{cases}$$

un diseño recursivo es inmediato:

```
int factorial(int x) {
   if (x==0) return 1;
   else return factorial(x-1)*x;
}
```

pero su definición iterativa es más eficaz:

```
int factorial(int x) {
  int f=1;
  for (i=1;i<=x;i++) f*=i;
  return f;
}
```

Las acciones no son recursivas o iterativas intrínsecamente: para toda acción recursiva existe una acción iterativa con su misma especificación, y viceversa. La recursividad se emplea libremente en la fase de diseño pero sólo se codifica recursivamente una acción cuando la versión iterativa equivalente es muy compleja.

En general, para resolver recursivamente un problema P sobre unos datos D el diseñador se pregunta si sería posible resolver P sobre los datos D suponiendo que ya está resuelto para otros datos D' del mismo tipo que D y en algún sentido más sencillos. Es decir, se trata de encontrar una relación de recurrencia en los datos del problema que permita calcular la solución pedida recurriendo a la solución para datos más simples. Es importante que en este planteamiento D sea del mismo tipo que D' y que, en un sentido bien definido, D' sea más pequeño que D.

Por ejemplo, supongamos que el problema P consiste en calcular a^n, donde los datos son el par $D = (a, n)$: $P((a, n)) = a^n$. El planteamiento recursivo consiste en resolver P sobre unos datos D' más pequeños que D. En este caso podemos elegir $D' = (a, n-1)$: si la solucion de P sobre D' es conocida ($P(D') = a^{n-1}$), la solución de P sobre D es $P(D) = a \cdot P(D')$.

Este razonamiento que conduce a calcular P sobre D en función de P sobre D' puede aplicarse para resolver P sobre D' en función de unos datos aun más pequeños D'', y así sucesivamente. Se forma entonces una sucesi ón $D > D' > D'' > \cdots$ de datos cada vez más pequeños. Para que el razonamiento sea correcto se requiere que la sucesión sea finita. Se llegará entonces a unos datos D_t lo

suficientemente pequeños como para que P pueda ser resuelto directamente sin recurrir a soluciones de P para otros datos. Diremos entonces que D_t es un caso trivial (en el ejemplo anterior, $D_t = (a, 0)$ y $P(D_t) = 1$). Cuando D no es lo suficientemente simple, diremos que es un caso recursivo o no trivial.

El aspecto de un programa recursivo es un reflejo de este análisis: habrá una o más instrucciones condicionales dedicadas a separar los tratamientos correspondientes a los casos triviales de los tratamientos correspondientes a los casos no triviales. Los primeros tienen el aspecto de un programa convencional mientras que en los segundos se pueden distinguir las siguientes partes:

- Primero se calculará el sucesor D' de D. También diremos que se produce una descomposición recursiva de los datos D para obtener los datos más sencillos D'. A veces nos referiremos a D' diciendo que es un subproblema de D.
- A continuación se produce una llamada recursiva para obtener la solución del subproblema D'.
- Finalmente, se opera sobre los resultados obtenidos para D' a fin de calcular la solución para D.

La solución C++ es la mostrada en el siguiente ejemplo:

```
int potencia(int a, int n) {
   if (n==0) return 1;
   else return a*potencia(a,n-1);
}
```

La condicion `n==0` identifica el caso trivial, cuya solución se conoce. En el caso no trivial la función ha de recurrir al resultado de aplicar la funcion a datos más pequeños. Toda función recursiva tiene al menos un caso trivial.

4

Definición de Nuevos Tipos de Datos

4.1. Definiciones de Tipos

Al comienzo del curso se ha visto que un programa C++ consistía en una combinación (secuencial, condicional o iterativa) de acciones de asignación, entrada y salida. Las acciones operan sobre datos, cuya representación en la memoria del ordenador se llamó objeto de almacenamiento, o simplemente "objeto" (ver sección 2.1). Cada objeto pertenece a un tipo, y un tipo es un conjunto de valores.

También se ha estudiado cómo extender el léxico del procesador con nuevas acciones, funciones y operadores. Estas, a su vez, eran composiciones de acciones primitivas y de otras acciones definidas con anterioridad. Sin excepción, cada uno de los objetos manejados por todos los programas que se han escrito hasta ahora pertenecía a uno de los tipos elementales predefinidos (carácter, entero, real, booleano) o era una tabla.

En esta lección se estudiará que también es posible definir conjuntos de objetos no pertenecientes a los tipos predefinidos. Esos conjuntos de valores son *tipos de datos definidos por el usuario* y extienden los tipos de datos *predefinidos* que se han utilizado hasta este momento. Se verá también que, al igual que las nuevas acciones eran equivalentes a una composición de acciones primitivas, los valores de los nuevos tipos se representarán mediante estructuras compuestas de valores pertenecientes a los tipos predefinidos.

4.2. Representación de nuevos tipos de datos

La forma más elemental de definir nuevos tipos consiste en enumerar sus componentes. Un tipo definido así no es distinto de un entero, porque siempre podrá reemplazarse cada valor del tipo por su ordinal. Una definición más flexible es la de tupla, que es una colección de objetos de cualquier tipo. Ambas se estudiarán en esta sección.

4.2.1. Definiciones de sinónimos: Orden “typedef”

Los nombres de los tipos pueden hacerse muy largos y poco legibles. La orden `typedef` sirve para crear un sinónimo del nombre de un tipo. Esta orden no crea un tipo nuevo, solamente da un nombre más al tipo que se desee.

Una declaración de variable precedida de la palabra `typedef` declara este nuevo nombre para el tipo en vez de una nueva variable del tipo dado. Por ejemplo,

```
typedef unsigned int uint;   // uint = unsigned int
typedef vector<uint> tabla;  // tabla = vector<unsigned int>
tabla a(10,1);               // vector<unsigned int> a(10,1);
uint b=4;                    // lo mismo que unsigned int b=4;
unsigned int c=4;
a[3]=b;
a[4]=c;
typedef vector<tabla> matriz;
matriz d(10,tabla(8));      // vector<vector<unsigned int> > d(10,
                            //       vector<unsigned int>(8));
d[2][4]=b;
```

4.2.2. Tipos enumerados: Tipo “enum”

Una enumeración es un tipo que puede almacenar un conjunto de valores especificados por el usuario. Por ejemplo, el conjunto de valores “semana” es {lunes, martes, etc. }. La definición de un tipo enumerado consiste en la palabra “enum” seguida por la lista de valores que componen el tipo, encerrados entre dos llaves y separados por comas. Los identificadores de los valores no pueden coincidir con los nombres de otro objeto, función o acción.

```
enum { lunes, martes, miercoles, jueves,
        viernes, sabado, domingo }
```

La declaración de una variable llamada “a” y de ese tipo es:

```
enum { lunes, martes, miercoles, jueves,
        viernes, sabado, domingo } a;
enum { a, b, c } d; // Incorrecto. 'a' es un nombre de variable
```

Las enumeraciones pueden tener nombre. Puede escribirse

```
enum semana { lunes, martes, miercoles, jueves,
        viernes, sabado, domingo };
semana a;
```

que es lo mismo que

```
typedef enum { lunes, martes, miercoles, jueves,
        viernes, sabado, domingo } semana;
semana a;
```

En el lenguaje C++ el procesador reemplaza cada uno de los símbolos por un número, empezando en cero. Por ejemplo, la enumeración siguiente

```
enum { lunes, martes, miercoles }
```

define la constante `lunes`, con valor 0, la constante `martes`, con valor 1 y la constante `miercoles` con valor 2.

Todas las variables de tipo enumerado en C++ se codifican como variables de tipo entero. Puede indicarse el entero que se desea asociar a uno o más de los símbolos:

```
enum e1 { oscuro, luminoso };     // oscuro=0, luminoso=1
enum e2 { a=3, b };               // a=3, b=4
enum e3 { min= -10, max=1000000 };
```

Un entero puede ser convertido explícitamente a un tipo enumerado, aunque el resultado de dicha conversión no está definido a menos que el valor esté en el rango correcto. Por ejemplo,

```
e1 x1 = e1(0)      // x1 es oscuro
e2 x3 = e2(3)      // x3 es a
e2 x4 = e2(5)      // Indefinido. 5 no está en e2
```

Las enumeraciones en C++ pueden manejarse con los mismos operadores que los enteros. Cuando se escribe en pantalla el valor de un tipo enumerado se obtiene el entero al que está asociado.

```
x1=oscuro;
cout << x1 << endl; // Se muestra en pantalla un cero
```

4.2.3. Tuplas: orden "struct"

Un `struct` es un agregado de elementos de tipos arbitrarios. Por ejemplo, los valores de la variable 'a', que se define a continuación, son tuplas de entero × vector de caracteres × entero:

```
struct {
      int DNI;
      vector<char> nombre;
      int edad;
} a;
```

Las estructuras pueden tener nombre, al igual que las enumeraciones:

```
struct direccion {
      int DNI;
      vector<char> nombre;
      int edad;
};
direccion a;
```

Los miembros individuales de cada variable se acceden mediante un punto (.). Por ejemplo,

```
direccion a;
a.DNI=11423545;
a.edad=23;
a.nombre=vector<char>(20);
a.nombre[2]='A';
```

Los objetos de tipo `struct` pueden ser asignados, pasados como argumentos a una función y devueltos como resultado de una función. Otras operaciones, como la comparación (==, !=) no están definidas, aunque el usuario puede programar los operadores correspondientes.

Aunque todos los elementos de la estructura sean del mismo tipo, una estructura no es un vector, como se muestra en el siguiente ejemplo:

```
struct T1 { int a, b, c; }
typedef vector<int> T2;
T1 v1;
T2 v2(3);
v1.a=1;  v1.b=2; v1.c=3;     // Correcto
v2[0]=1; v2[1]=2; v3[2]=3;   // Correcto
v1=v2;                       // INCORRECTO
v1[0]=1;                     // INCORRECTO
```

Se considera que dos estructuras son diferentes incluso si tienen los mismos miembros:

```
struct A { int a; char b; };
struct B { int a; char b; };
int main() {
  A x; x.a=10; x.b='a';
  B y;
  y=x;     // Error: x e y son de tipos diferentes
  y.a=x.a; // Correcto
  y.b=x.b;
}
```

Normalmente no se declaran estructuras dentro de una función. Aunque en el ejemplo anterior pudo haberse escrito

```
int main() {
  struct A { int a; char b; };
  struct B { int a; char b; };
  A x; x.a=10; x.b='a';
  B y;
  y.a=x.a;
  y.b=x.b;
}
```

en otros casos que se estudiarán más adelante la definición de una estructura dentro de una función produce un error de compilación.

Punteros a estructuras

Si pd es una variable de tipo "puntero a estructura" la operación (*pd).miembro puede abreviarse en pd->miembro:

```
direccion *p;
p=new direccion;      // Ver sección 4.5
(*p).edad=23;
p->edad=23;           // La misma asignación que antes
delete p;
```

4.3. Tipos Abstractos de Datos

Todos los tipos tienen asociado un conjunto de acciones, funciones y operaciones que actúan sobre ellos. Por ejemplo, la suma, la resta y la multiplicación son algunas de las operaciones asociadas al tipo "entero". De la misma forma, existirán acciones, funciones y operaciones asociadas a cualquiera de los tipos definidos por el usuario.

Los valores de los tipos predefinidos se codificaban mediante cadenas de bits, para poder almacenarse en la memoria del ordenador. Pero es posible escribir la mayoría de los programas sin conocer exactamente la forma en que un objeto está representado. Por ejemplo, no es necesario conocer la codificación en complemento a dos para programar una suma, porque el código del operador "+" es coherente con esa representación y no necesitamos saber cómo está programado ese operador para utilizarlo. Es suficiente con conocer las propiedades de la suma, que se definen de forma independiente de la representación.

En general, en el diseño de un programa no siempre es importante conocer de qué forma se representan los valores de cada tipo. Pero sí lo es conocer **las propiedades de las operaciones definidas sobre él**. Siguiendo con el ejemplo de los números enteros, el dígito 7 es un símbolo asociado a un elemento del conjunto de los números naturales. Los romanos habrían asignado el símbolo VII al mismo elemento, y el ordenador emplea la cadena 00000000 00000000 00000000 00000111. ¿Cuál es, entonces, la definición del conjunto de los números naturales? La enumeración de los símbolos con que se representa cada elemento en la numeración decimal, $\mathbf{N} = \{0, 1, 2, 3, 4, 5, \ldots\}$, es tan arbitraria como la enumeración de los símbolos con que se representa en la numeración romana, $\mathbf{N} = \{I, II, III, IV, V, \ldots\}$ o en la numeración binaria que emplea el ordenador. Un tipo puede definirse sin asociar un símbolo a cada uno de sus elementos si se enumeran las propiedades de las operaciones que están asociadas a sus valores:

Ejemplo: El conjunto **N** de los números naturales se puede definir como sigue:

1. 0 es un número natural
2. Si x es un número natural, entonces su sucesor $s(x)$ es también un número natural.
3. 0 no es el sucesor de ningún número natural.
4. Si x e y son diferentes, entonces $s(x)$ y $s(y)$ son diferentes.

5. Si una propiedad P es cierta para el 0, y si P es cierta para $s(x)$ siempre que P es cierta para x, entonces P es cierta para todos los números naturales.

Las expresiones 0, s(0), s(s(0)), etc. hacen referencia a los mismos elementos que son representados por los dígitos 0, 1, 2, etc. Esta definición se conoce como *axiomática de Peano* de los números naturales. Para especificar la operación "+" uno puede decir:

$$x + 0 = x \quad \forall x \in \mathbf{N}$$
$$x + s(y) = s(x + y) \quad \forall x, y \in \mathbf{N}$$

con lo que quedan definidas las propiedades de la suma, no importa cual sea la reprentación. Por ejemplo,

$$3 + 2 = 3 + s(1) = s(3 + 1) = s(3 + s(0)) = s(s(3 + 0)) = s(s(3)) = s(4) = 5$$
$$III + II = III + s(I) = s(III + I) = s(III + s(0)) = s(s(III + 0)) = s(s(III)) = s(IV) = V$$
$$s(s(s(0))) + s(s(0)) = s(s(s(s(0))) + s(0)) = s(s(s(s(s(0))) + 0)) = s(s(s(s(s(0)))))$$

Distinguiremos entre la **especificación** y la **implementación** de un tipo de datos. Diremos que la **especificación** de un tipo de datos consiste en definir las propiedades de las operaciones definidas sobre él. Por el contrario, la **implementación** de un tipo consiste en elegir una representación del tipo en función de los tipos predefinidos, y en implementar sus operaciones de forma que cumplan su especificación.

Existen técnicas formales para especificar un tipo de datos (se han empleado en el ejemplo para especificar la suma), aunque no las estudiaremos en este curso. Nosotros utilizaremos la palabra "especificación" para referirnos a un texto en el que se expliquen las propiedades que tienen que cumplir las operaciones que actúan sobre los elementos de un tipo definido por el usuario.

4.3.1. Definiciones

La **encapsulación de datos** (también llamada **ocultación de información**) consiste en ocultar los detalles de la implementación de un tipo de datos a sus usuarios.

La **abstracción de datos** es la separación entre la **especificación** de un tipo de datos y su **implementación**.

Un **Tipo Abstracto de Datos** es un tipo de datos organizado de forma que la especificación de los valores y la especificación de las operaciones sobre los valores están separadas de la representación de los valores y de la implementación de las operaciones.

4.4. Implementación de tipos abstractos de datos: Orden "class"

El lenguaje C++ proporciona un mecanismo para permitir la distinción entre la especificación y la implementación y para ocultar la implementación de un tipo abstracto de datos de sus usuarios. La orden *class*, que generaliza a la orden *struct* que se ha estudiado antes, se empleará para implementar TADs.

Al igual que *struct*, la orden *class* sirve para definir un tipo de datos “tupla” y las operaciones definidas sobre él. Consta de cuatro partes:

- El nombre de la clase
- Los campos que definen la tupla, también llamados **datos miembro** o **atributos**.
- Las operaciones que se definen sobre el tipo, también llamadas **funciones miembro**
- Los niveles de acceso desde el programa, que controlan en grado de acceso de las funciones definidas fuera de la clase a los miembros (datos y funciones) de ésta. Hay tres niveles de acceso: **public**, **protected** y **private**. El nivel de acceso *public* significa que el miembro es accesible desde cualquier función, el nivel *private* significa que el miembro sólamente puede ser accedido desde otra función miembro de la misma clase. El nivel *protected* se estudiará en el capítulo 6.

4.4.1. Funciones miembro

Las operaciones que afectan a valores del tipo que se define en una *struct* o *class* se pueden declarar o definir dentro de la estructura o clase. La definición de una función miembro consiste en mover el código dentro de la definición de la clase, como se muestra en el siguiente ejemplo.

Ejemplo: Definición del tipo “número racional” y el operador “suma de racionales”.

En este ejemplo definimos el tipo `Q` sin ocultación de información ni abstracción de datos (es decir, Q **no** es la implementación del TAD “Q”).

```
#include <iostream>
using namespace std;
struct Q {
   int num;
   unsigned int den;
};

void swap(int &a, int &b) {
   // Intercambia los valores de dos variables enteras
   int c=a; a=b; b=c;
}

int mcd(int a, int b) {
   // Calcula el máximo común divisor de dos enteros
   do {
     if (a<b) swap(a,b);
     if (b==0) return a;
     a=a-b;
   } while(b!=0);
```

```
}
int mcm(int a, int b) {
   // Calcula el mínimo común múltiplo de dos enteros
   return a*b/mcd(a,b);
}

Q simplifica(const Q &x) {
   // Calcula una fracción irreducible
   Q resultado;
   int m=mcd(x.num,x.den);
   resultado.num=x.num/m;
   resultado.den=x.den/m;
   return resultado;
}

Q operator+(const Q &x, const Q &y) {
   // Suma de números racionales
   Q resultado;
   resultado.den=mcm(x.den,y.den);
   resultado.num=x.num*resultado.den/x.den+y.num*resultado.den/y.den;
   return simplifica(resultado);
}

int main() {
   Q a,b,c;
   cout << "Dame numerador y denominador de a: ";
   // Se accede a los datos miembro ‘‘num’’ y ‘‘den’’
   // luego no hay ocultación de información
   cin >> a.num >> a.den;
   cout << "Dame numerador y denominador de b: ";
   cin >> b.num >> b.den;
   c=a+b;
   cout << "a+b=" << c.num << "/" << c.den << endl;
}
```

Para convertir a la función *simplifica* en una función miembro, se puede mover su código dentro de la definición de la estructura:

```
struct Q {
   int num;
   unsigned int den;
   Q simplifica(const Q &x) {
     Q resultado;
     int m=mcd(x.num,x.den);
     resultado.num=x.num/m;
     resultado.den=x.den/m;
     return resultado;
   }
};
```

o bien declararla dentro de la estructura y definirla fuera. Se emplea la sintaxis

```
tipo-retorno nombre_clase :: nombre_miembro ( argumentos );
```

que es muy similar a la que se usa en los *namespace*, como se ve a continuación:

```
struct Q {
   int num;
   unsigned int den;
   Q simplifica(const Q &x);
};

Q Q::simplifica(const Q &x) {
   Q resultado;
   int m=mcd(x.num,x.den);
   resultado.num=x.num/m;
   resultado.den=x.den/m;
   return resultado;
}
```

En el lenguaje C++ todas las funciones, acciones y operadores miembro tienen un argumento oculto llamado **this**, que es un puntero a la clase a la que pertenecen. En ocasiones se dice que “una función miembro conoce el objeto desde el que fue llamado”. En realidad, en una de las fases de la compilación se transforma la declaración de la función *simplifica* del modo que sigue:

```
   Q Q::simplifica(const Q &x) --->
        Q simplifica(const Q *this,const Q &x)
```

y las llamadas a esta función se transforman así:

```
   resultado.simplifica(x) --->
       simplifica(&resultado,x);
```

Este argumento oculto puede emplearse dentro de las funciones miembro. Por ejemplo, podemos definir la acción `simplifica` de la forma siguiente:

```
void Q::simplifica(void) {
   int m=mcd(num,den);
   num/=m;
   den/=m;
}
```

que es convertida automáticamente por el compilador en la siguiente definición:

```
void Q::simplifica(void) {
   int m=mcd(this->num,this->den);
   this->num/=m;
   this->den/=m;
}
```

En el ejemplo siguiente se usa esta notación en la función simplifica, que pasa a ser una acción y se define de la misma manera el operador “+”:

```
#include <iostream>
using namespace std;
struct Q {
   int num;
   unsigned int den;
   void simplifica(void);
   Q operator+(const Q &x);
   void construye(int n, int d);
   void escribe();
};

void swap(int &a, int &b) {
   int c=a; a=b; b=c;
}

int mcd(int a, int b) {
   do {
     if (a<b) swap(a,b);
     if (b==0) return a;
     a=a-b;
   } while(b!=0);
}
int mcm(int a, int b) {
   return a*b/mcd(a,b);
}

void Q::simplifica(void) {
   int m=mcd(num,den);
   num/=m;
   den/=m;
}

Q Q::operator+(const Q &x) {
   Q resultado;
   resultado.den=mcm(x.den,den);
   resultado.num=x.num*resultado.den/x.den+num*resultado.den/den;
   resultado.simplifica();
   return resultado;
}

void Q::construye(int n, int d) {
   num=n; den=d;
}

void Q::escribe() {
   cout << num << "/" << den << endl;
}
```

```
int main() {
   Q a,b,c; int n,d;
   cout << "Dame numerador y denominador de a: ";
   cin >> n >> d; a.construye(n,d);
   cout << "Dame numerador y denominador de b: ";
   cin >> n >> d; b.construye(n,d);
   c=a+b;
   cout << "a+b=";
   c.escribe();
   cout  << endl;
}
```

La sintaxis de la definición del operador “+” no es intuitiva. La definición

```
Q Q::operator+(const Q &x) {
   Q resultado;
   resultado.den=mcm(x.den,den);
   resultado.num=x.num*resultado.den/x.den+num*resultado.den/den;
   resultado.simplifica();
   return resultado;
}
```

es convertida automáticamente en

```
Q operator+(const Q *this, const Q &x) {
   Q resultado;
   resultado.den=mcm(x.den,this->den);
   resultado.num=x.num*resultado.den/x.den+this->num*resultado.den/this->den;
   resultado.simplifica();
   return resultado;
}
```

con lo que la llamada

```
c=a+b;
```

en convertida en primer lugar a

```
c=a.operator+(b);
```

que, a su vez es reemplazada por

```
c=operator+(*a,b);
```

es decir, el objeto del que el operador es miembro es el primer argumento del operador.

4.4.2. Miembros públicos y privados

Para implementar un TAD es necesario ocultar la representación del tipo, empleando las palabras *public* y *private*. Estas van seguidas de dos puntos e indican que, a partir del punto en que se encuentran, todos los miembros son públicos o

privados, respectivamente. Todos los miembros de una *class* son privados, y todos los miembros de una *struct* son públicos salvo que se indique lo contrario. Por ejemplo, las siguientes definiciones son equivalentes:

```
class direccion {
     public:
     int DNI;
     vector<char> nombre;
     int edad;
};
```

```
struct direccion {

   int DNI;
   vector<char> nombre;
   int edad;
};
```

En el ejemplo que sigue, los miembros *DNI* y *edad* son privados, y *nombre* es público.

```
class direccion {
     int DNI;
     public:
     vector<char> nombre;
     private:
     int edad;
};
```

Siguiendo con el ejemplo de la sección anterior, puede hacerse que los datos miembros "*num*" y "*den*" del tipo racional y la función "*simplifica*" sean privados, reemplazando la estructura por una clase, de la forma siguiente:

```
class Q {
   int num;
   unsigned int den;
   void simplifica(void);
   public:
   Q operator+(const Q &x);
};
```

Si se hace esto, en la función *main* hay que reemplazar las líneas en que se consultan o modifican *num* y *den*. El compilador daría un error si tratásemos de consultar o modificar esos datos miembro:

```
int main() {
   Q a,b,c;
   cout << "Dame numerador y denominador de a: ";
   cin >> a.num >> a.den;                          // ERROR
   cout << "Dame numerador y denominador de b: ";
   cin >> b.num >> b.den;                          // ERROR
   c=a+b;
   cout << "a+b=" << c.num << "/" << c.den << endl; // ERROR
}
```

La forma más sencilla de facilitar el acceso desde funciones no miembro requiere definir varias funciones miembro públicas, como "construye" y "escribe" en el programa que sigue:

```
#include <iostream>
using namespace std;
class Q {
   // Implementación del TAD ‘‘Q’’
   int num;
   unsigned int den;
   void simplifica(void);
   public:
   Q operator+(const Q &x);
   void construye(int n, int d);
   void escribe();
};

void swap(int &a, int &b) {
   int c=a; a=b; b=c;
}

int mcd(int a, int b) {
   do {
     if (a<b) swap(a,b);
     if (b==0) return a;
     a=a-b;
   } while(b!=0);
}
int mcm(int a, int b) {
   return a*b/mcd(a,b);
}

void Q::simplifica(void) {
   int m=mcd(num,den);
   num/=m;
   den/=m;
}

Q Q::operator+(const Q &x) {
   Q resultado;
   resultado.den=mcm(x.den,den);
   resultado.num=x.num*resultado.den/x.den+num*resultado.den/den;
   resultado.simplifica();
   return resultado;
}

void Q::construye(int n, int d) {
   num=n; den=d;
}

void Q::escribe() {
   cout << num << "/" << den << endl;
```

```
}

int main() {
   Q a,b,c; int n,d;
   cout << "Dame numerador y denominador de a: ";
   cin >> n >> d; a.construye(n,d);
   cout << "Dame numerador y denominador de b: ";
   cin >> n >> d; b.construye(n,d);
   c=a+b;
   cout << "a+b=";
   c.escribe();
   cout  << endl;
}
```

En esta última forma de programar el algoritmo la codificación de un número racional está oculta, luego está de acuerdo con la definición de TAD. A continuación se estudia otro ejemplo en el que se ve más clara la importancia práctica de ocultar la representación de un TAD.

Días entre dos fechas

Supongamos que deseamos programar un algoritmo que nos devuelva el número de días entre dos fechas del siglo XX. Para ello contamos el número de días entre el 1 de enero de ambas fechas, teniendo en cuenta cuántos años bisiestos hay, y a continuación contamos el número de días que hay entre el uno de enero de cada uno de los años y las fechas dadas. El operador "−" se redefine de forma que, cuando sus argumentos sean fechas, devuelva el número de días entre ellas.

El resultado, sin ocultación de información ni abstracción de datos, se muestra a continuación. Después se irán haciendo cambios, hasta llegar a la implementación del TAD "fecha":

```
#include <iostream>
using namespace std;

struct fecha {
    int dia;
    int mes;
    int anho;
};

bool bisiesto(int anho) {
  if (anho%4!=0) return false;
  if (anho%100==0) {
   if (anho%4000==0) return true; else return false;
  }
  return true;
}

int operator-(const fecha &f1, const fecha &f2) {
```

```
 vector<int> diasmes(12,31);
 diasmes[1]=28; diasmes[3]=30; diasmes[5]=30;
 diasmes[8]=30; diasmes[10]=30;

 int i, dia0=0, dia1=0, dia2=0;

 // Número de días entre el 1 de enero de f1 y f2
 for (i=f1.anho;i<f2.anho;i++)
   if (bisiesto(i)) dia0+=366; else dia0+=365;

 // Número de días desde el 1 de enero, fecha 1
 for (i=0;i<f1.mes-1;i++) dia1+=diasmes[i];
 dia1+=f1.dia-1;
 if (f1.mes>=2 && bisiesto(f1.anho)) dia1++;

 // Número de días desde el 1 de enero, fecha 2
 for (i=0;i<f2.mes-1;i++) dia2+=diasmes[i];
 dia2+=f2.dia-1;
 if (f2.mes>=2 && bisiesto(f2.anho)) dia2++;

 return dia2-dia1+dia0;

}

int main() {

   fecha f1, f2;
   int distancia;

   cout << "Dime día, mes y año de la primera fecha: ";
   cin >> f1.dia >> f1.mes >> f1.anho;

   cout << "Dime día, mes y año de la segunda fecha: ";
   cin >> f2.dia >> f2.mes >> f2.anho;

   distancia=f1-f2;

   cout << "El número de días es " << distancia << endl;
}
```

En la siguiente versión hacemos que el operador “−” sea miembro de *fecha*. Esto será necesario si se desea que el código de este operador acceda a la representación del tipo “fecha” cuando, más adelante, declaremos los miembros “dia”, “mes”, “anho” como privados:

```
#include <iostream>
using namespace std;
```

```
struct fecha {
    int dia;
    int mes;
    int anho;
    int operator-(const fecha &f2);
};

bool bisiesto(int anho) {
  if (anho%4!=0) return false;
  if (anho%100==0) {
   if (anho%4000==0) return true; else return false;
  }
  return true;
}

int fecha::operator-(const fecha &f2) {
 vector<int> diasmes(12,31);
 diasmes[1]=28; diasmes[3]=30; diasmes[5]=30;
 diasmes[8]=30; diasmes[10]=30;

 int i, dia0=0, dia1=0, dia2=0;

 // Número de días entre el 1 de enero de f1 y f2
 for (i=anho;i<f2.anho;i++)
   if (bisiesto(i)) dia0+=366; else dia0+=365;

 // Número de días desde el 1 de enero, fecha 1
 for (i=0;i<mes-1;i++) dia1+=diasmes[i];
 dia1+=dia-1;
 if (mes>=2 && bisiesto(anho)) dia1++;

 // Número de días desde el 1 de enero, fecha 2
 for (i=0;i<f2.mes-1;i++) dia2+=diasmes[i];
 dia2+=f2.dia-1;
 if (f2.mes>=2 && bisiesto(f2.anho)) dia2++;

 return dia2-dia1+dia0;

}

int main() {

   fecha f1, f2;
   int distancia;

   cout << "Dime día, mes y año de la primera fecha: ";
   cin >> f1.dia >> f1.mes >> f1.anho;
```

```
   cout << "Dime día, mes y año de la segunda fecha: ";
   cin >> f2.dia >> f2.mes >> f2.anho;

   distancia=f1-f2;

   cout << "El número de días es " << distancia << endl;
}
```

Por último, se oculta la representación de la fecha, haciendo que los datos miembro sean privados:

```
#include <iostream>
using namespace std;

class fecha {
    // TAD ‘‘fecha’’, versión 1
    int dia;
    int mes;
    int anho;
    public:
    void construye(int d, int m, int a);
    int operator-(const fecha &f2);
};

bool bisiesto(int anho) {
  if (anho%4!=0) return false;
  if (anho%100==0) {
   if (anho%4000==0) return true; else return false;
  }
  return true;
}

void fecha::construye(int d, int m, int a) {
    dia=d; mes=m; anho=a;
}

int fecha::operator-(const fecha &f2) {
 vector<int> diasmes(12,31);
 diasmes[1]=28; diasmes[3]=30; diasmes[5]=30;
 diasmes[8]=30; diasmes[10]=30;

 int i, dia0=0, dia1=0, dia2=0;

 // Número de días entre el 1 de enero de f1 y f2
 for (i=anho;i<f2.anho;i++)
   if (bisiesto(i)) dia0+=366; else dia0+=365;

 // Número de días desde el 1 de enero, fecha 1
 for (i=0;i<mes-1;i++) dia1+=diasmes[i];
```

```
 dia1+=dia-1;
 if (mes>=2 && bisiesto(anho)) dia1++;

 // Número de días desde el 1 de enero, fecha 2
 for (i=0;i<f2.mes-1;i++) dia2+=diasmes[i];
 dia2+=f2.dia-1;
 if (f2.mes>=2 && bisiesto(f2.anho)) dia2++;

 return dia2-dia1+dia0;

}

int main() {

   int d,m,a,distancia;
   fecha f1,f2;

   cout << "Dime día, mes y año de la primera fecha: ";
   cin >> d >> m >> a; f1.construye(d,m,a);

   cout << "Dime día, mes y año de la segunda fecha: ";
   cin >> d >> m >> a; f2.construye(d,m,a);

   distancia=f1-f2;

   cout << "El número de días es " << distancia << endl;
}
```

Ahora programaremos otra implementación diferente del mismo TAD "fecha". En todos los programas anteriores la fecha se codifica mediante un triplete (día, mes, año). Pero si el objetivo es calcular diferencias entre fechas, será más eficiente emplear un único entero: el número de días transcurridos desde el 1 de enero del año 1900. En la siguiente versión se ve cómo las funciones en que se usa un TAD (en este caso, la función *main*) no deben cambiar si, como en este caso, se decide cambiar la representación del TAD sin alterar las declaraciones de las funciones miembro públicas (hay quien llama a estas declaraciones públicas el **interface** del TAD).

```
#include <iostream>
using namespace std;

class fecha {
    // TAD ``fecha'', versión 2.
    // La representación se modifica
    int ndias;
    public:
    // Esta parte no cambia
    void construye(int d, int m, int a);
    int operator-(const fecha &f2);
```

```
};

bool bisiesto(int anho) {
  if (anho%4!=0) return false;
  if (anho%100==0) {
   if (anho%4000==0) return true; else return false;
  }
  return true;
}

void fecha::construye(int d, int m, int a) {

 vector<int> diasmes(12,31);
 diasmes[1]=28; diasmes[3]=30; diasmes[5]=30;
 diasmes[8]=30; diasmes[10]=30;
 int i, dias=0;

 // Número de días entre 1-1-1900 y 1-1-anho
 for (i=1900;i<a;i++)
   if (bisiesto(i)) dias+=366; else dias+=365;

 // Número de días desde el 1 de enero
 for (i=0;i<m-1;i++) dias+=diasmes[i];
 dias+=d-1; if (m>=2 && bisiesto(a)) dias++;

 ndias=dias;

}

int fecha::operator-(const fecha &f2) {

  return f2.ndias-ndias;

}

int main() {

   int d,m,a,distancia; fecha f1,f2;

   cout << "Dime día, mes y año de la primera fecha: ";
   cin >> d >> m >> a; f1.construye(d,m,a);

   cout << "Dime día, mes y año de la segunda fecha: ";
   cin >> d >> m >> a; f2.construye(d,m,a);

   distancia=f1-f2;

   cout << "El número de días es " << distancia << endl;
```

```
}
```

4.5. Extensión temporal de la definición de una variable

Como se ha mencionado en la seccion 3.7, un objeto declarado en una función se crea cuando se encuentra su definición y se destruye cuando su nombre sale del ámbito. Tales objetos son llamados **automáticos**. Los objetos declarados en el ámbito global o en un módulo (`namespace`) y los declarados `static` dentro de una función se inician solamente una vez y viven hasta que el programa termina. Estos objetos se llaman **estáticos**. Ahora veremos que también es posible controlar la creación y la destrucción de objetos mediante las órdenes `new` y `delete`.

El operador `new` tiene como argumento un nombre de tipo y devuelve un puntero que contiene la dirección de una posición en la memoria que no está ocupada por ninguna variable. Se dice que un objeto creado por `new` está en el **almacenamiento libre** o **memoria dinámica**.

El operador `delete` tiene como argumento un puntero a una variable dinámica y libera el espacio que esta variable ocupa para que pueda ser empleado por otras aplicaciones de `new`. Por ejemplo:

```
struct pareja {
      re, im: float;
};

void main(void) {
     pareja *p;     // p contiene una dirección cualquiera de memoria
     p=new pareja;  // p contiene una dirección de memoria no utilizada
     (*p).re=3.0;   // utilizamos esa memoria
     p->im=1.0;     // lo mismo, con otra notación

     // ...

     delete p;      // liberamos la memoria
}
```

4.6. Constructores de tipos

En los ejemplos de implementación de TADs se ha definido una acción miembro "construye", que tiene como argumentos datos de otros tipos y que modifica los datos miembro privados de un objeto. Este problema se plantea siempre que se implementa un TAD, y por esta razón en el lenguaje C++ se proporciona una sintaxis más cómoda para dar el valor inicial a los datos miembro privados de un objeto. Esta consiste en definir una acción miembro con una sintaxis especial: tiene el mismo nombre que el tipo que se está definiendo y no tiene tipo de retorno. Esta acción se denomina **constructor** del tipo, o simplemente constructor.

Por ejemplo, en el TAD "Q" se programó la clase `Q` de esta forma:

```
class Q {
   int num;
   unsigned int den;
   void simplifica(void);
   public:
   Q operator+(const Q &x);
   void construye(int n, int d);
   void escribe();
};

void Q::construye(int n, int d) {
   num=n; den=d;
}
```

Si se emplea un constructor, la sintaxis es:

```
class Q {
   int num;
   unsigned int den;
   void simplifica(void);
   public:
   Q operator+(const Q &x);
   Q(int n, int d); // En vez de ''void construye(int n, int d);''
   void escribe();
};

Q::Q(int n, int d) {
   num=n; den=d;
}
```

También hay diferencias en la forma en que se llama a la función. En el ejemplo anterior se escribió

```
Q a; int n,d; cin >> n >> d; a.construye(n,d);
```

Si se emplea un constructor, hay dos alternativas:

```
int n,d; cin >> n >> d; Q a(n,d);
Q a; int n,d; cin >> n >> d; a=Q(n,d);
```

Con la expresión `Q a(n,d);` se crea una variable de tipo `Q` llamada `a`, y se llama al constructor con los argumentos `n` y `d`. En la secuencia `Q a; a=Q(n,d);` primero se crea una variable llamada `a`, de tipo `Q`. Después, en la asignación `a=Q(n,d);` se crea un objeto de tipo `Q`, *sin nombre*, y se llama al constructor con los argumentos `n` y `d`. Después se copia el contenido de los datos miembros del objeto sin nombre a los datos miembros del objeto `a`. Más adelante se verá cómo redefinir el significado del operador de asignación (sección 4.8).

Es posible sobrecargar el constructor de un objeto. Esto es, puede haber dos acciones miembro que se llamen igual que el TAD y con argumentos de tipos distintos. Se elige uno u otro de acuerdo con la forma en que se invoque. Por ejemplo, en la siguiente definición puede construirse un número racional a partir de una pareja de enteros o de un único entero:

```
class Q {
   int num;
   unsigned int den;
   void simplifica(void);
   public:
   Q operator+(const Q &x);
   Q(int n, int d);
   Q(int n);
   void escribe();
};

Q::Q(int n, int d) {
   num=n; den=d;
}
Q::Q(int n) {
   num=n; den=1;
}
```

4.7. Destructores

Algunas veces, la creación de un objeto emplea un recurso, generalmente la reserva de memoria dinámica. El `destructor` de una clase es una función que se llama cuando el objeto se destruye (las reglas por las que un objeto perteneciente a una clase se destruye son las mismas que para cualquier otra variable: salir de ámbito si es automática, llamada a `delete` si es dinámica, final del programa si es global). El destructor tiene el mismo nombre que el constructor, precedido de una vírgula (˜) y no tiene argumentos.

Ejemplo:

```
#include <iostream>
using namespace std;
struct datos {
   int a, b,c;
};

class estructura_dinamica {
   datos *A;
   public:
   estructura_dinamica(int a, int b, int c) {
       A=new datos;
       A->a=a; A->b=b; A->c=c;
       cout << "Objeto creado" << endl;
   }
   ~estructura_dinamica() {
       delete A;
       cout << "Objeto destruido" << endl;
```

```
   }
   int suma() { return A->a+A->b+A->c; }
};

void f(int n) {
   int i;
   estructura_dinamica a(1,2,3); // Se crea: llamada al constructor
   cout << n+a.suma() << endl;
   // Se destruye: llamada al destructor
}

void main(void) {
     f(2);
     f(3);
}
```

El programa anterior muestra en pantalla el siguiente texto:

```
Objeto creado
8
Objeto destruido
Objeto creado
9
Objeto destruido
```

4.8. Asignación y paso por valor de objetos como argumentos

Cada variable de una clase es una representación de un tipo abstracto de datos en una tupla de valores. Esto es, el contenido de la variable está compuesto por los valores de los datos definidos dentro de la clase. Por tanto, la asignación consiste en la copia de esos valores.

En la clase `Q` definida en la sección 4.4 los campos eran `num` y `den`, de tipo entero. Una asignación copiará los valores de ambos datos y generará un objeto correcto. Pero en la clase `mi_vector`, definida en la sección 4.7, los campos eran `A`, de tipo puntero a entero, y `n`, de tipo entero. La asignación copiará el valor de `A`, que es la dirección en que está el contenido del vector, de un objeto a otro, lo que es incorrecto. La asignación debería copiar el contenido del vector.

El mismo problema aparece en el paso por valor de la variable como argumento a una función, porque es necesario construir un nuevo objeto, copia del que se está pasando. El parámetro formal se inicializa llamando a un **constructor de copia** del tipo. El constructor de copia tiene como argumento una referencia al propio tipo (es decir, el constructor de copia de un tipo T será, en general, una acción "`T(const T&)`"). En el paso por referencia, donde no se pasaría el objeto sino un nombre alternativo para él, no se usa el constructor de copia.

En los casos en que la asignación por defecto y la construcción por defecto a partir de un elemento de la misma clase son inadecuadas es necesario redefinir este constructor y la operación de asignación, como se ve en el ejemplo que sigue:

Ejemplo:

```
#include <iostream>
using namespace std;
struct datos {
   int a, b,c;
};

class estructura_dinamica {
   datos *A;
   public:

   estructura_dinamica(int a, int b, int c) {
       A=new datos;
       A->a=a; A->b=b; A->c=c;
       cout << "En el constructor a partir de enteros" << endl;
   }

   ~estructura_dinamica() {
       delete A;
       cout << "En el destructor" << endl;
   }

   estructura_dinamica() {
       // Constructor por defecto
       A=new datos;
       A->a=0; A->b=0; A->c=0;
       cout << "En el constructor por defecto" << endl;
   }

   estructura_dinamica(const estructura_dinamica &a) {
       A=new datos;
       A->a=(a.A)->a; A->b=(a.A)->b; A->c=(a.A)->c;
       cout << "En el constructor de copia" << endl;
   }

   void operator=(const estructura_dinamica &a) {
       A->a=a.A->a; A->b=a.A->b; A->c=a.A->c;
       cout << "En el operador de asignación" << endl;
   }

   int suma() { return A->a+A->b+A->c; }
};

void f_valor(estructura_dinamica a) {
   cout << a.suma() << endl;
}
void f_referencia(estructura_dinamica &a) {
```

```
   cout << a.suma() << endl;
}

void main(void) {
     estructura_dinamica a(1,2,3);
     estructura_dinamica b;
     b=a;
     f_valor(b);
     f_referencia(b);
}
```

Este programa muestra en pantalla el siguiente resultado:

```
En el constructor a partir de enteros
En el constructor por defecto
En el operador de asignación
En el constructor de copia
6
En el destructor
6
En el destructor
En el destructor
```

4.9. Tipos parametrizados: Orden “template”

El lenguaje C++ permite crear **tipos paramétricos**, (esto ya se ha visto en la construcción “vector de T”, donde “T” es un tipo cualquiera). La notación es como sigue:

```
template <typename T> class nombre_clase { ... };
```

o bien

```
template <typename T> struct nombre_clase { ... };
```

El nombre del tipo resultante se compone del nombre de la clase y del tipo que se pasa como parámetro entre símbolos “< >”:

```
nombre_clase<X>
```

4.9.1. Acciones genéricas

Si un argumento de una acción (también función u operador) pertenece a un tipo paramétrico, se dice que la acción es genérica. Por ejemplo:

```
#include <iostream>
using namespace std;

template<typename T> void swap(T &a, T&b) {
   T c=a; a=b; b=c;
}

int main() {
  int a1=1, b1=2;
  swap(a1,b1);
  cout << a1 << " " << b1 << endl;
  float a2=1, b2=2;
  swap(a2,b2);
  cout << a2 << " " << b2 << endl;
}
```

En el ejemplo que sigue se definen dos clases paramétricas y dos acciones genéricas:

```
#include <iostream>
using namespace std;

template <typename T> struct datos {

   // Tipo paramétrico

   T a, b,c;
};

template <typename T> class estructura_dinamica {

   // Tipo paramétrico

   datos<T> *A;
   public:

   estructura_dinamica(T a, T b, T c) {
       A=new datos<T>;
       A->a=a; A->b=b; A->c=c;
       cout << "En el constructor a partir de enteros" << endl;
   }

   ~estructura_dinamica() {
       delete A;
       cout << "En el destructor" << endl;
   }

   estructura_dinamica() {
       // Constructor por defecto
```

```
        A=new datos<T>;
        A->a=0; A->b=0; A->c=0;
        cout << "En el constructor por defecto" << endl;
    }

    estructura_dinamica(const estructura_dinamica &a) {
        A=new datos<T>;
        A->a=(a.A)->a; A->b=(a.A)->b; A->c=(a.A)->c;
        cout << "En el constructor de copia" << endl;
    }

    void operator=(const estructura_dinamica &a) {
        A->a=a.A->a; A->b=a.A->b; A->c=a.A->c;
        cout << "En el operador de asignación" << endl;
    }

    T suma() { return A->a+A->b+A->c; }
};

template <typename T> void f_valor(estructura_dinamica<T> a) {
    // Acción genérica
    cout << a.suma() << endl;
}

template <typename T> void f_referencia(estructura_dinamica<T> &a) {
    // Acción genérica
    cout << a.suma() << endl;
}

void main(void) {
     estructura_dinamica<int> a(1,2,3);
     estructura_dinamica<int> b;
     estructura_dinamica<float> c(1.5,2.5,3.5);
     estructura_dinamica<float> d;
     b=a; d=c;
     f_valor(b);
     f_referencia(b);
     f_valor(d);
     f_referencia(d);
}
```

El programa anterior muestra en pantalla el siguiente resultado:

```
En el constructor a partir de enteros
En el constructor por defecto
En el constructor a partir de enteros
En el constructor por defecto
En el operador de asignación
En el operador de asignación
```

```
En el constructor de copia
6
En el destructor
6
En el constructor de copia
7.5
En el destructor
7.5
En el destructor
En el destructor
En el destructor
En el destructor
```

4.10. Objetos como miembros de otras clases

En el siguiente ejemplo se repite el programa visto en la sección anterior sin emplear memoria dinámica. En la definición de `estructura_automatica`, la clase `datos` es miembro de `estructura_automatica` (en `estructura_dinámica` se usó un puntero). En este nuevo caso no es necesario utilizar destructores. Obsérvese que se han utilizado **inicializadores**, o llamadas al constructor de la clase miembro desde el constructor de la clase que lo contiene. Los inicializadores se escriben detras de la declaración del constructor, separados de él por dos puntos (:) y por comas entre sí, como se verá en el siguiente ejemplo.

```
#include <iostream>
using namespace std;

template <typename T> class datos {

   // Tipo paramétrico

   T a,b,c;
   public:
   datos() {
     a=0; b=0; c=0;
     cout << "En constructor por defecto de datos" << endl;
   }
   datos(const datos &d) {
     a=d.a; b=d.b; c=d.c;
     cout << "En constructor de copia de datos" << endl;
   }
   datos(T pa, T pb, T pc) {
     a=pa; b=pb; c=pc;
     cout << "En constructor desde enteros de datos" << endl;
   }
   void operator=(const datos &d) {
     a=d.a; b=d.b; c=d.c;
```

```
    cout << "En el operador de asignación de datos" << endl;
  }
  T suma() { return a+b+c; }

};

template <typename T> class estructura_automatica {

  // Tipo paramétrico

  datos<T> A;
  public:

  estructura_automatica(T a, T b, T c) : A(a,b,c) {
      cout << "En el constructor a partir de enteros de e.a." << endl;
  }

  estructura_automatica() : A() {
      cout << "En el constructor por defecto de e.a." << endl;
  }

  estructura_automatica(const estructura_automatica &a) : A(a.A) {
      cout << "En el constructor de copia de e.a." << endl;
  }

  void operator=(const estructura_automatica &a) {
      A=a.A;
      cout << "En el operador de asignación de e.a." << endl;
  }

  T suma() { return A.suma(); }
};

template <typename T> void f_valor(estructura_automatica<T> a) {
  // Acción genérica
  cout << a.suma() << endl;
}

template <typename T> void f_referencia(estructura_automatica<T> &a) {
  // Acción genérica
  cout << a.suma() << endl;
}

void main(void) {
     estructura_automatica<int> a(1,2,3);
     estructura_automatica<int> b;
     estructura_automatica<float> c(1.5,2.5,3.5);
     estructura_automatica<float> d;
```

```
    b=a; d=c;
    f_valor(b);
    f_referencia(b);
    f_valor(d);
    f_referencia(d);
}
```

El programa anterior muestra en pantalla el siguiente resultado:

```
En constructor desde enteros de datos
En el constructor a partir de enteros de e.a.
En constructor por defecto de datos
En el constructor por defecto de e.a.
En constructor desde enteros de datos
En el constructor a partir de enteros de e.a.
En constructor por defecto de datos
En el constructor por defecto de e.a.
En el operador de asignación de datos
En el operador de asignación de e.a.
En el operador de asignación de datos
En el operador de asignación de e.a.
En constructor de copia de datos
En el constructor de copia de e.a.
6
6
En constructor de copia de datos
En el constructor de copia de e.a.
7.5
7.5
```

5

Estructuras de datos

En este capítulo se mostrará cómo utilizar algunos de los tipos abstractos de datos de uso más frecuente. Todos los tipos que se definirán son paramétricos. Por ejemplo, se verá que un elemento del tipo "vector de enteros" es una tupla de enteros, un elemento del tipo "vector de reales" es una tupla de reales y, en general, un elemento del tipo "vector de T", para cualquier tipo "T" es una tupla de valores de "T". Para referirnos a ese parámetro "T" usaremos la expresión "tipo base".

5.1. El tipo abstracto de datos "vector"

La estructura de datos más habitual es el vector (también llamada tabla o array). El tipo vector es el producto cartesiano de un tipo base por sí mismo n veces. De forma más general, un vector es un conjunto V de pares que cumple que para cada valor i $\in$ indice, existe un valor x y sólo uno tal que $(x, i) \in V$. Este conjunto puede representarse de varias formas:

5.1.1. Representación en posiciones contiguas

La representación más corriente del tipo vector consiste en un conjunto de posiciones de memoria consecutivas (ver figura 5.1). Esta representación es eficiente en tiempo, porque es muy sencillo acceder a cada elemento del vector, pero puede ser ineficiente en memoria, porque las casillas no ocupadas ocupan tanto espacio como las ocupadas.

Figura 5.1: Representación de un vector en posiciones contiguas de memoria

5.1.2. Representación de matrices dispersas

Una matriz **dispersa** es una matriz en la que la mayoría de los elementos son cero. Las matrices dispersas se almacenan enumerando los elementos no nulos, como una lista de triples

$$(i, j, valor)$$

mas el número de filas, columnas y elementos no nulos.

Por ejemplo, la matriz

$$A = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

se podría representar con una matriz de $3 \times N$ elementos, más el número de filas de la matriz, el número de columnas y el número de términos no nulos. Estas últimas tres variables suelen guardarse en la primera fila de la matriz. En este ejemplo, ésta sería

$$\begin{pmatrix} 6 & 6 & 8 \\ 1 & 1 & 15 \\ 1 & 4 & 22 \\ 1 & 6 & -15 \\ 2 & 2 & 11 \\ 2 & 3 & 3 \\ 3 & 4 & -6 \\ 5 & 1 & 91 \\ 6 & 3 & 28 \end{pmatrix}$$

Puede verse que si la matriz no es dispersa la representación mediante triples es más ineficiente que la contigua. Existen operaciones para trasponer y multiplicar matrices dispersas con una eficiencia poco menor que en matrices contiguas.

5.1.3. Vectores de librería STL

El nombre de la clase es `vector`, y su constructor tiene un argumento, la dimensión del vector. Un vector STL, de nombre "`x`", dimensión N y tipo base T, se declara así:

```
vector<T> x(N);
```

La función miembro `size()` devuelve la dimensión del vector:

```
void f(vector<int> u) {
   int i;
   for (i=0;i<u.size();i++) cout << u[i];
}

vector <int> x(100),y;
```

```
for (i=0;i<100;i++) x[i]=2*i;
y=x;
f(y);
```

Para crear un vector de objetos de una clase en la que no esté definido un constructor por defecto, es necesario proporcionar explícitamente el valor de cada elemento. Por esta razón el constructor del vector está sobrecargado y se puede invocar con dos argumentos: la dimensión del vector y el valor inicial de cada uno de sus elementos. Por ejemplo:

```
// ‘‘Num’’ es el tipo abstracto ‘‘entero
// de precisión arbitraria’’
class Num {
    public:
    Num(int);  // Se construye a partir de un
               // entero predefinido
    // No hay constructor por defecto
    // ...
};
vector<Num> v1(1000);        // ERROR
vector<Num> v2(1000,Num(0)); // Correcto
```

Los constructores de copia y de asignación copian los elementos de un vector. Para un vector con muchos elementos, esto puede ser lento, de modo que los vectores se pasan típicamente por referencia. Por ejemplo:

```
void f1(vector<int>&);        // Común
void f2(const vector<int>&);  // Común
void f3(vector<int>);         // Correcto, pero lento

void h() {
     vector<int> v(10000);

     // ...

     f1(v);    // Pasa una referencia
     f2(v);    // Pasa una referencia
     f3(v);    // Copia los 10000 elementos en un nuevo vector
}
```

5.2. El tipo abstracto de datos “pila”

Una pila es una lista en la que todas las inserciones y eliminaciones se hacen por el mismo extremo (ver figura 5.2).

El tipo pila se implementa mediante la clase `stack`. Sea `T` el tipo base de la pila, `p` una pila y `t` un valor del tipo base. Las funciones miembro de una pila son:

- Constructor `stack()`: produce una pila vacía.

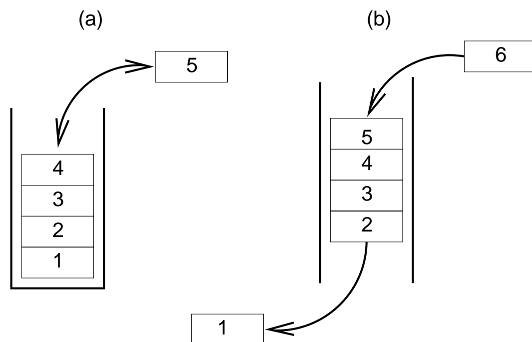

Figura 5.2: Pila (a) y cola (b). En la pila las inserciones y las eliminaciones se hacen por el mismo lado. En la cola se inserta por un extremo y se elimina por el otro

- `T stack<T>::top()`: Es una función que devuelve un elemento del tipo base, que es copia del elemento que se ha añadido en último lugar.
- `void stack<T>::push(const T &x)`: Añade una copia del elemento `x` a la pila.
- `void stack<T>::pop()`: Elimina el elemento que está en la parte superior de la pila.
- `bool stack<T>::empty()`: Devuelve `true` si la pila está vacía, y falso si no lo está.

Por ejemplo:

```
#include <iostream>
#include <stack>
using namespace std;
void main(void) {
    stack<int> q;
    int i;

    for (i=0;i<10;i++) q.push(i);
    while (!q.empty()) {
       cout << q.top();
       q.pop();
    }
}
```

5.3. El tipo abstracto de datos "cola"

Una cola es una lista en la que las inserciones se hacen por un extremo y las eliminaciones por el otro (ver figura 5.2).

El tipo cola se implementa mediante la clase `queue`. Sea `T` el tipo base de la cola, `c` una cola y `t` un valor del tipo base. Las funciones miembro de una cola son:

- Constructor `queue()`: produce una pila vacía.
- `T queue<T>::front()`: Devuelve una copia del elemento que se insertó en primer lugar en la cola.
- `void queue<T>::push(const T &x)`: Encola una copia del elemento `x`.
- `void queue<T>::pop()`: Elimina el primer elemento que se insertó en la cola.
- `bool queue<T>::empty()`: Devuelve `true` si la cola está vacía, y falso si no lo está.

Por ejemplo:

```
#include <iostream>
#include <queue>
using namespace std;
void main(void) {
    queue<int> q;
    int i;

    for (i=0;i<10;i++) q.push(i);
    while (!q.empty()) {
       cout << q.front();
       q.pop();
    }
}
```

5.4. El tipo abstracto de datos “cadena de caracteres”

Una cadena de caracteres se emplea para almacenar textos y consiste en un vector de tamaño variable de caracteres. Las cadenas de caracteres se implementan mediante la clase `string`. Estudiaremos las siguientes operaciones:

- Extracción del carácter que está en la posición *i*-ésima: De igual forma que si la variable fuese de tipo `vector`
- Comparación: Están definidos los operadores `< > == !=`. Una cadena es menor que otra si la precede en el orden alfabético. Las mayúsculas preceden a las minúsculas en el código ASCII.
- Concatenación: El operador + produce la concatenación de sus argumentos.

- Búsqueda de subcadenas: La función miembro `find` tiene como argumento la subcadena que se busca y devuelve la posición de comienzo de ésta, si se encuentra, y -1 si no.
- Extracción de subcadenas: La función miembro `substr` tiene como argumentos dos números, la posición del comienzo de la subcadena que se desea extraer y su longitud.

Por ejemplo, el siguiente programa produce el resultado que se muestra a continuación:

```
#include <string>
#include <iostream>
using namespace std;
int main(void) {
   string s1="Hola",s2;
   cout << s1 << endl;

   // Asignación
   s2="abcdefghijklmnopqrstuvwxyz";

   // Carácter en posición 7
   cout << "El carácter que está en la posición 7 es " << s2[7] << endl;

   // Comparación (según el orden del código ASCII)
   if (s1<s2) cout << s1 << " precede a " << s2 << endl;
   else cout << s2 << " precede a " << s1 << endl;

   // Concatenación
   string s3=s1+s2;
   cout << s3 << endl;

   // Búsqueda de subcadenas
   int pos;
   pos=s2.find("mnq");
   cout << "La posición de mnq es " << pos << endl;
   pos=s2.find("mno");
   cout << "La posición de mno es " << pos << endl;

   // Extracción de subcadenas
   s3=s2.substr(pos,3);
   cout << "La subcadena es " << s3 << endl;

   // Longitud de la cadena
   cout << "La longitud de s3 es " << s3.size() << endl;

}
$ a.out
Hola
```

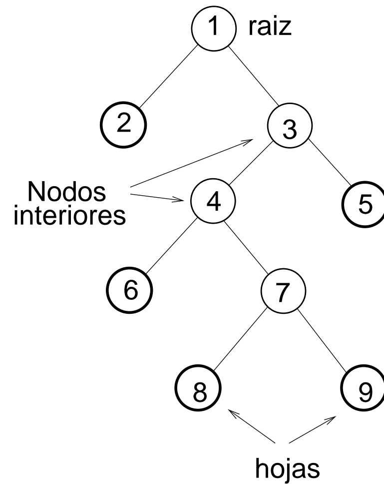

Figura 5.3: Un árbol binario no vacío consta de una raíz y dos subárboles, izquierdo y derecho. Los nodos que no tienen subárboles se llaman nodos terminales u hojas. Los demás nodos se llaman nodos interiores.

```
El carácter que está en la posición 7 es h
Hola precede a abcdefghijklmnopqrstuvwxyz
Holaabcdefghijklmnopqrstuvwxyz
La posición de mnq es -1
La posición de mno es 12
La subcadena es mno
La longitud de s3 es 3
```

5.5. El tipo abstracto de datos “arbol binario”

Un **árbol binario** es un conjunto finito de nodos que está vacío o bien consiste en una raíz y en dos árboles binarios disjuntos llamados subárbol izquierdo y subárbol derecho (ver figura 5.3).

Los árboles se usan principalmente para almacenar información que no esté indexada por números. Por ejemplo, suponga que desea almacenar en memoria una lista con las notas de todos los alumnos matriculados en la asignatura de informática. Para ello puede emplearse un vector de registros. Si se desea conocer la nota de un alumno, dado su primer apellido, es necesario recorrer uno por uno todos los nombres de la tabla hasta encontrar una coincidencia, de la forma siguiente:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct alumno {
   string nombre;
```

```
   string apellido1;
   string apellido2;
   float nota;
};

int main() {
   int nalumnos;
   cout << "Cuántos alumnos hay? "; cin >> nalumnos;
   vector<alumno> tabla(nalumnos);
   for (int i=0;i<tabla.size();i++) {
      cout << "Alumno número " << i << endl;
      cout << "Nombre y apellidos del alumno: ";
      cin >> tabla[i].nombre;
      cin >> tabla[i].apellido1;
      cin >> tabla[i].apellido2;
      cout << "Nota: "; cin >> tabla[i].nota;
   }

   string consulta;
   cout << "Dime el primer apellido:"; cin >> consulta;
   for (int i=0;i<tabla.size();i++) {
      if (consulta==tabla[i].nombre) {
        cout << "La nota es " << tabla[i].nota << endl;
      }
   }

}
```

El esquema basado en el vector no es eficiente si el número de alumnos es muy elevado. Imagine ahora que la información se guarda de la forma que se muestra en la figura 5.4. Cada nodo interno del árbol es una pregunta que separa en dos mitades la parte de la lista que queda por buscar, de forma que el número medio de nodos que hay que visitar cuando se busca un alumno es de $\log_2$(tamaño de tabla).

Este tipo de árboles de búsqueda se programa en C++ mediante la clase `map`. La representación interna en forma de árbol está oculta, y las variables de este tipo se utilizan como si fuesen un tipo especial de vectores. En el `map` es posible emplear un tipo distinto del entero para indexar un elemento, como se muestra a continuación:

```
#include <iostream>
#include <string>
#include <vector>
#include <map>
using namespace std;

struct alumno {
   string nombre;
   string apellido1;
```

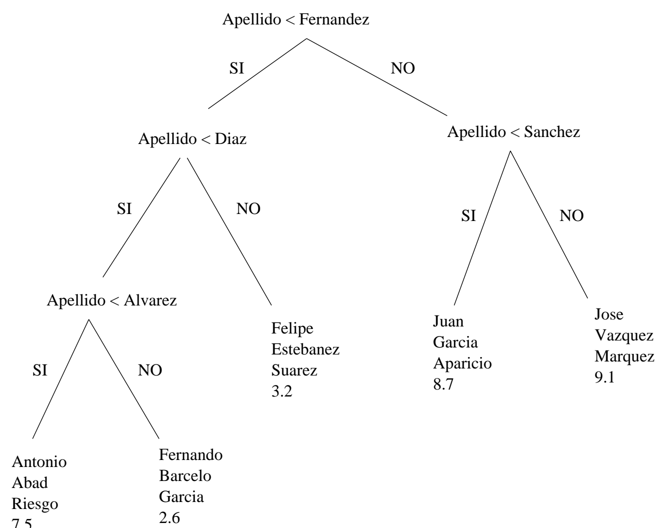

Figura 5.4: Forma de almacenar una tabla de nombres y notas en un árbol binario

```
   string apellido2;
   float nota;
};

int main() {
   int nalumnos;
   cout << "Cuántos alumnos hay? "; cin >> nalumnos;
   map<string,alumno> tabla;
   alumno x;
   for (int i=0;i<nalumnos;i++) {
      cout << "Alumno número " << i << endl;
      cout << "Nombre y apellidos del alumno: ";
      cin >> x.nombre >> x.apellido1 >> x.apellido2;
      cout << "Nota: "; cin >> x.nota;
      tabla[x.apellido1]=x;
   }

   string consulta;
   cout << "Dame el primer apellido: "; cin >> consulta;
   cout << tabla[consulta].nota;
}
```

El requisito mínimo que debe cumplir un tipo para poder ser empleado como índice un `map` es el de tener definida la operación de comparación entre sus elementos. Las comparaciones entre `string` están definidas en la STL y por eso se pudo emplear sin más. En el siguiente ejemplo se muestra cómo emplear un tipo cuya operación de comparación está definida por el usuario:

```
#include <iostream>
```

```
#include <string>
#include <vector>
#include <map>
using namespace std;

struct alumno {
   string nombre;
   string apellido1;
   string apellido2;
};

bool operator<(const alumno &a1, const alumno &a2) {
   if (a1.apellido1>a2.apellido1) return false;
   if (a1.apellido2>a2.apellido2) return false;
   if (a1.nombre>a2.nombre) return false;
   return true;
}

int main() {
   int nalumnos; float nota;
   cout << "Cuántos alumnos hay? "; cin >> nalumnos;
   map<alumno,float> tabla;
   alumno x;
   for (int i=0;i<nalumnos;i++) {
      cout << "Alumno número " << i << endl;
      cout << "Nombre y apellidos del alumno: ";
      cin >> x.nombre >> x.apellido1 >> x.apellido2;
      cout << "Nota: "; cin >> nota;
      tabla[x]=nota;
   }

   alumno consulta;
   cout << "Dame los datos del alumno: ";
   cin >> consulta.nombre >> consulta.apellido1 >> consulta.apellido2;
   cout << tabla[consulta];
}
```

Existen otras operaciones definidas sobre el tipo `map`. Puede consultarse si un elemento está o no en el árbol mediante la función `find` y es posible recorrer en orden todos los valores almacenados. Se incluye un ejemplo de cómo hacer ambas cosas, aunque la explicación de su funcionamiento está fuera de los objetivos de este curso.

```
#include <iostream>
#include <string>
#include <vector>
#include <map>
using namespace std;
```

```
struct alumno {
   string nombre;
   string apellido1;
   string apellido2;
};

bool operator<(const alumno &a1, const alumno &a2) {
   if (a1.apellido1>a2.apellido1) return false;
   if (a1.apellido2>a2.apellido2) return false;
   if (a1.nombre>a2.nombre) return false;
   return true;
}

int main() {
   int nalumnos; float nota;
   cout << "Cuántos alumnos hay? "; cin >> nalumnos;
   map<alumno,float> tabla;
   alumno x;
   for (int i=0;i<nalumnos;i++) {
      cout << "Alumno número " << i << endl;
      cout << "Nombre y apellidos del alumno: ";
      cin >> x.nombre >> x.apellido1 >> x.apellido2;
      cout << "Nota: "; cin >> nota;
      tabla[x]=nota;
   }

   alumno consulta;
   cout << "Dame los datos del alumno: ";
   cin >> consulta.nombre >> consulta.apellido1 >> consulta.apellido2;
   if (tabla.find(consulta)!=tabla.end()) cout << tabla[consulta] << endl;
   else cout << "Alumno inexistente" << endl;

   map<alumno,float>::iterator px;
   for (px=tabla.begin();px!=tabla.end();px++) {
        x=px->first;
        nota=px->second;
        cout << x.nombre << " " << x.apellido1 << " " << x.apellido2 << " ";
        cout << nota << endl;
   }

}
```

5.6. El tipo abstracto de datos “grafo”

De manera informal, un grafo es la abstracción de un mapa de carreteras. Los elementos que componen cada valor se corresponden con las ciudades del mapa y las conexiones entre ellos con las carreteras. Los grafos tienen muchas aplicaciones

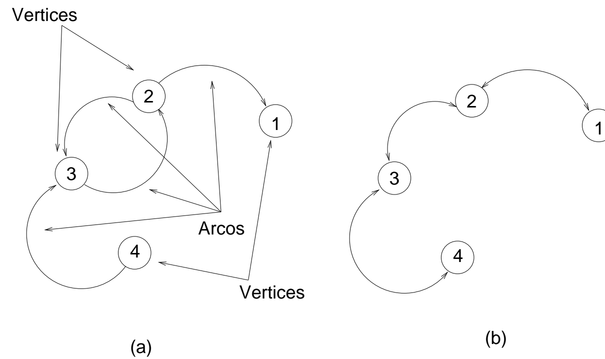

Figura 5.5: Grafo dirigido (a) y no dirigido (b). Un grafo consta de un conjunto V de vértices y de un conjunto de arcos A, donde $A \subset V \times V$. En un grafo no dirigido si $(u, v) \in A$ entonces $(v, u) \in A$.

prácticas. Por ejemplo, los simuladores de circuitos eléctricos, de hidráulica o de estructuras se programan mediante grafos.

5.6.1. Definiciones

Un grafo es una estructura compuesta por un conjunto de vértices V y un conjunto A de arcos (también llamados "aristas") que unen esos vértices. Se denota por (u, v) al arco que sale del vértice $u \in V$ y llega al vértice $v \in V$, por lo que $A \subset V \times V$.

Distinguiremos entre grafos dirigidos y no dirigidos (ver figura 5.5). En un grafo no dirigido los pares no están ordenados y el arco (u, v) es el mismo que el arco (v, u). Si (v_1, v_2) es un arco de A, entonces diremos que los vértices v_1 y v_2 son **adyacentes** y que el arco (v_1, v_2) es incidente en los vértices v_1 y v_2.

Un **subgrafo** de G es un grafo G' tal que $V(G') \subset V(G)$ y $A(G') \subset A(G)$. Un **camino** desde el vértice v_p hasta el vértice v_q en un grafo G es una secuencia de vértices $v_p, v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_q$ tal que (v_p, v_{i_1}),(v_{i_1}, v_{i_2}),..., (v_{i_n}, v_q) son arcos de $A(G)$. Un **camino simple** es un camino en el que todos los vértices excepto posiblemente el primero y el último son distintos. Si el camino es (1,2)(2,4)(4,3) se escribe 1,2,4,3. Los caminos 1,2,4,3 y 1,2,4,2 tienen **longitud** 3, el primero de ellos es simple y el segundo no. Un **ciclo** es un camino en el que el primer y el último vértice son el mismo. Un grafo sin ciclos se llama **acíclico**. En grafos dirigidos se añade la palabra "dirigido" a los términos ciclo y camino. En un grafo no dirigido G dos vértices están conectados si existe un camino en G de v_1 a v_2. Un grafo no dirigido G está **conectado** si existen caminos entre cualquier par de sus vértices (un árbol es un grafo acíclico conectado).

5.6.2. Recorridos de un grafo

El **recorrido en profundidad** de un grafo opera de la forma siguiente: se visita el primer vértice v. A continuación, se selecciona un vértice w adyacente a v y se inicia una nueva búsqueda en profundidad. Cuando se alcanza un vértice u para el que todos sus vértices adyacentes hayan sido visitados, se termina esa búsqueda

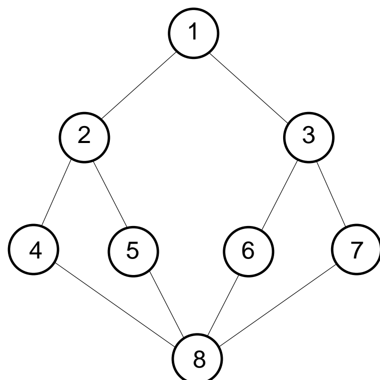

Figura 5.6: Grafo no dirigido

y se vuelve al último vértice visitado que tiene un vértice adyacente w adyacente a él y se inicia una nueva búsqueda lineal desde w. El programa termina cuando no puede alcanzarse ningún vértice no visitado desde alguno de los visitados.

La forma más sencilla de describir este esquema es recursiva (su implementación C++ está más adelante, en esta misma sección). Si el grafo de la figura 5.6 se recorre en profundidad, sus vértices se procesarán en el orden

1 3 7 8 6 2 5 4

En el **recorrido en anchura**, todos los vértices adyacentes a v que no hayan sido visitados son recorridos en primer lugar. A continuación se recorren los vértices adyacentes a éstos que no hayan sido visitados, y así sucesivamente. Si el grafo de la figura 5.6 se recorre en anchura, sus vértices se recorren en el orden

1 3 2 7 6 5 4 8

A continuación se incluye una implementación en C++ del tipo abstracto "grafo". Un grafo consiste en dos conjuntos, de vértices y aristas, que se han representado mediante un vector de elementos `V`, que almacena los datos asociados a los vértices, y una matriz de booleanos `A`, que almacena las aristas. El elemento `A[i][j]` es `true` si hay un arco que sale del vértice almacenado en la posición `i` de la tabla `V` y llega al vértice numerado con `j` en la misma tabla. A esta matriz se la conoce como **matriz de adyacencias**. Hay dos funciones miembro, `anade_vertice` y `anade_arista`, que añaden a los conjuntos V y A nuevos elementos. La función `adyacentes` devuelve una pila con los vértices adyacentes a uno dado.

```
#include <vector>
#include <iostream>
#include <string>
#include <stack>
using namespace std;

template <typename T> class grafo {
```

```
vector<T> V;
vector<vector<bool> > A;
int nvertices;
public:
 grafo(int maxv) {
    V=vector<T>(maxv);
    A=vector<vector<bool> >(maxv);
    for (int i=0;i<A.size();i++) A[i]=vector<bool>(maxv);
    nvertices=0;
 }
 void anade_vertice(const T &x) {
   // No se inserta si ya está
   for (int i=0;i<nvertices;i++) if (x==V[i]) return;

   // Se inserta el vértice
   V[nvertices]=x;
   nvertices++;
 }
 void anade_arista(const T &x, const T &y) {

   // Se comprueba que existan los vértices
   int lugar1=-1,lugar2=-1;
   for (int i=0;i<nvertices;i++)  if (x==V[i]) { lugar1=i; break; }
   for (int i=0;i<nvertices;i++)  if (y==V[i]) { lugar2=i; break; }

   if (lugar1==-1 || lugar2==-1) return; // No existen

   // Se comprueba que no exista la arista
   if (A[lugar1][lugar2]) return;

   // Se inserta la arista
   A[lugar1][lugar2]=true;
 }

 T operator[](const int i) { return V[i]; }

 int nvert() { return nvertices; }

 stack<T> adyacentes(const T &x) const {

   stack<T> result; int posicion=-1;
   for (int i=0;i<nvertices;i++) if (x==V[i]) { posicion=i; break; }

   // Si no existe, cola vacía
   if (posicion==-1) return result;

   // Existe: se almacenan todos los vértices adyacentes
   for (int i=0;i<V.size();i++)
```

```
        if (A[posicion][i]) result.push(V[i]);

     return result;

   }

};


template <typename T> void concatena(stack<T> &s1, stack<T> s2) {
  // Concatena dos pilas
  stack<T> s3;
  while (!s1.empty()) { s3.push(s1.top()); s1.pop(); }
  while (!s2.empty()) { s3.push(s2.top()); s2.pop(); }
  while (!s3.empty()) { s1.push(s3.top()); s3.pop(); }
}

template <typename T> bool pertenece(const T &x, stack<T> s) {
  // Cierto si un elemento pertenece a una pila
  while (!s.empty()) { if (s.top()==x) return true; s.pop(); }
  return false;
}

template <typename T> void anchura( const grafo<T> &g,
                                    const T &primero,
                                    void (*P)(const T&)) {
    stack<T> visitados;
    stack<T> faltan;

    (*P)(primero);
    visitados.push(primero);
    concatena(faltan,g.adyacentes(primero));
    while (!faltan.empty()) {
        T elemento=faltan.top();
        faltan.pop();
        if (!pertenece(elemento,visitados)) {
            visitados.push(elemento);
            (*P)(elemento);
            concatena(faltan,g.adyacentes(elemento));
        }
    }

}


template <typename T> void tmp_prof(const grafo<T> &g,
                                    const T &primero,
                                    void (*P)(const T&),
```

```
                                    stack<T> &visitados) {

    (*P)(primero); visitados.push(primero);
    stack <T> ady=g.adyacentes(primero);
    while (!ady.empty()) {
        T elemento=ady.top(); ady.pop();
        if (!pertenece(elemento,visitados)) tmp_prof(g,elemento,P,visitados);
    }
}

template <typename T> void profundidad( const grafo<T> &g,
                                    const T &primero,
                                    void (*P)(const T&)) {
   stack<T> visitados;
   tmp_prof(g,primero,P,visitados);


}

void escribe(const string &s) {
   cout << s << endl;
}

int main() {
     int maxv;
     cout << "Número máximo de vértices? "; cin >> maxv;
     grafo<string> ejemplo(maxv);

     for (int i=0;i<maxv;i++) {
        string palabra;
        cout << "Teclea una palabra: "; cin >> palabra;
        ejemplo.anade_vertice(palabra);
     }
     for (int i=0;i<ejemplo.nvert();i++) {
        for (int j=0;j<ejemplo.nvert();j++) {
          cout << "Arco (" << ejemplo[i] << "," << ejemplo[j] << ") (s/n): ";
          char resp; cin >> resp;
          if (resp=='s' || resp=='S')
             ejemplo.anade_arista(ejemplo[i],ejemplo[j]);
        }
     }

     cout << "Recorrido en anchura " << endl;
     anchura(ejemplo,ejemplo[0],escribe);
     cout << "Recorrido en profundidad " << endl;
     profundidad(ejemplo,ejemplo[0],escribe);

}
```

5.7. El tipo abstracto “fichero secuencial”

El tipo abstracto **fichero** en C++ es similar al tipo `string`. Ambos son cadenas de caracteres de longitud arbitraria. Las diferencias entre un fichero y un `string` están en su representación. El contenido de un fichero no se almacena en la memoria electrónica del ordenador, sino en un dispositivo externo, en general en un disco duro, de mayor capacidad pero de acceso más lento. Esto significa que el contenido de cada variable de tipo fichero en C++ está almacenado a su vez en un archivo, gestionado por el sistema operativo que se esté utilizando, lo que permite conservar el contenido de esta variable después de que el programa termine.

Los ficheros se emplean en lugar de los strings en dos casos:

1. La longitud de la cadena es demasiado grande como para almacenarla en la memoria del computador.
2. Los datos que contiene la cadena deben perdurar más tiempo del que el programa está activo. Por ejemplo, en un fichero pueden guardarse los datos personales de los empleados de una empresa o el resultado de invertir una matriz de 10000 elementos.

Distinguiremos dos tipos de fichero: los **ficheros de entrada** y los **ficheros de salida**. Los ficheros de entrada sirven para que un programa lea datos desde ellos y los ficheros de salida para que el programa escriba datos en ellos. Los ficheros de entrada se implementan en C++ mediante la clase `ifstream` y los ficheros de salida mediante la clase `ofstream`.

Es útil imaginarse que los ficheros se almacenan en una cinta dividida en casillas, cada una de las cuales contiene un carácter. En un fichero de escritura hay operaciones para borrar una cinta y para escribir un carácter en la siguiente posición libre. En un fichero de lectura hay operaciones para posicionarse en la primera casilla de una cinta, para leer el siguiente carácter de una cinta y para preguntar si quedan más caracteres en la cinta.

5.7.1. Clase ofstream

Las operaciones que pueden realizarse sobre la clase `ofstream` son las siguientes:

- Borrado de una cinta: En el constructor del objeto se indica el nombre del archivo asociado al `ofstream`. Al crearse el objeto, el contenido de ese fichero se vacía.
- Escritura de un carácter en la siguiente posición libre: La función miembro `put` almacena el carácter que se le pasa como argumento a continuación del último que se ha escrito (es similar al miembro `push` de una cola)

Ejemplo:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
   ofstream f("datos.dat");
   string s="Esta frase se repetirá mil veces";
   for (int i=0;i<1000;i++) {
     for (int j=0;j<s.size();j++) f.put(s[j]);
   }
}
```

5.7.2. Clase ifstream

Las operaciones que pueden realizarse sobre la clase `ifstream` son las siguientes:

- Posicionarse en la primera posición de una cinta: En el constructor del objeto se le indica, como se hizo en el `ofstream`, el nombre del archivo que contiene la lista de caracteres. La primera operación de lectura devolverá el primer carácter contenido en el archivo.
- Lectura de un carácter: La función miembro `get` devuelve el siguiente carácter de la cinta (es similar a la función `pop` de una cola).
- Preguntar si quedan más caracteres en una cinta: La operación de comparación de un objeto de tipo `ifstream` con un número entero está definida de forma que si no quedan más caracteres en el archivo la expresión `nombre-objeto == 0` es cierta.

Ejemplo:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
   // Escribe en pantalla el contenido de un fichero
   ifstream f("datos.dat");
   while (f) {
      char c=f.get();
      cout << c;
   }
}
```

5.7.3. Entrada y salida con formato

Si se desea almacenar el valor de una variable en un fichero, ésta ha de representarse mediante una secuencia de caracteres. Por ejemplo, para almacenar un número entero `n` se representa el número como una lista de dígitos y se escriben uno a uno:

```
   int reves=0, resto=n;
   while (resto>0) {
      reves=reves*10+resto%10;
      resto=resto/10;
   }
   while (reves>0) {
       f.put(char(reves%10+int('0')));
       reves=reves/10;
   }
```

Obsérvese que ésta es precisamente la operación que se realiza mediante el operador << cuando se escribe se escribe en pantalla una variable de tipo entero: `cout << i;` . Si `n` es una variable de tipo entero, real o carácter y `g` es el nombre de una variable de tipo `ofstream`, la expresión

```
g << n;        // operator<<(g,n)
```

almacena la lista de caracteres correspondiente en `g`.

El tipo de retorno del operador $<<$ es “referencia a fichero de salida”, de forma que tienen sentido expresiones como

```
g << n << m; // operator<<(operator<<(g,n),m)
```

El operador $>>$ hace la función opuesta a partir de un fichero de lectura. La orden

```
f >> n;       // operator>>(f,n)
```

lee varios dígitos del fichero `f` y compone con ellos un entero. La lectura ignora los espacios en blanco (y tabuladores, saltos de página y cambios de línea) hasta encontrar el primer dígito y termina correctamente cuando se encuentra un nuevo espacio en blanco, carácter de fin de línea o tras leer el último carácter del fichero, o bien termina con un error si se encuentra un carácter incorrecto.

El número de decimales que se produce al convertir un `float` o `double` a una secuencia de caracteres se controla mediante una llamada a la función miembro `precision(n)`, que controla el número de cifras significativas del número. El modificador `width(n)` especifica el mínimo número de caracteres que se emplearán en la siguiente operación $<<$:

```
#include <iostream>
using namespace std;
void main(void) {
    cout.precision(4);
    cout.width(6);
    cout << 1.123456789 << "\n";
    cout.width(12);
    cout << 1.123456789 << "\n";
    cout.precision(20);
    cout << 1.123456789 << "\n";
}
```

`cout` y `cin` son dos objetos de tipos de `ofstream` y `ifstream`. Los datos enviados a `cout` se envían a la pantalla del ordenador y los datos con los que opera `cin` se leen del teclado.

Los operadores << y >> pueden sobrecargarse para escribir a un ofstream y a un ifstream, como se muestra en el ejemplo que sigue:

```
#include <iostream>
#include <fstream>
#include <vector>
using namepsace std;

template <typename T> ostream & operator<<(ostream &os, vector<T> &x) {
   for (int i=0;i<x.size();i++) os << x[i] << " ";
   return os;
}

template <typename T> istream & operator>>(istream &is, vector<T> &x) {
   for (int i=0;i<x.size();i++) is >> x[i];
   return is;
}

int main() {
    vector<float> a(10);
    cin >> a;
    cout << a;
    ofstream f("datos.dat");
    f << a;
}
```

6

Jerarquías de Tipos

6.1. Subtipado. Herencia y Jerarquías de tipos

Un tipo D' es un **subtipo** de un **supertipo** D si todos los elementos de D' también pertenecen a D. Por ejemplo, los enteros son un subtipo de los reales porque todos los enteros son, a su vez, números reales. Los reales son también un subtipo de los complejos. Cualquier función definida en D también es aplicable a los objetos del subtipo D': la suma de números complejos es aplicable a la suma de números reales y enteros. Por el contrario, puede haber funciones definidas en el subtipo que no sean aplicables al supertipo, cómo el modulo de dos números, que sólo está definido si ambos son enteros. En un lenguaje más formal, si f es una aplicación de D en otro conjunto X, entonces los pares $\{(d, f(d) \mid d \in D'\}$ son una aplicación de D' en X, pero una aplicación definida en D' no necesariamente es una aplicación cuando el conjunto original se extiende a D.

En los lenguajes de programación modernos se le permite al programador especificar relaciones entre supertipos y subtipos y definir:

- Una operación que sea aplicable a todos los subtipos, pero que se programe de formas diferentes en cada uno. Por ejemplo, un tipo `matriz` con dos subtipos: matrices contiguas y matrices dispersas. Las operaciones definidas sobre una matriz, como la suma o la traspuesta, tienen que ser aplicables a todos los tipos de matrices. Pero la implementación más eficiente depende de la representación: no se usa el mismo algoritmo para trasponer un matriz dispersa que para trasponer una matriz rectangular.

- Una operación aplicable a un subtipo pero no al supertipo. Por ejemplo, el resto de dividir un número entre otro tiene sentido en los números enteros, pero no en los números reales. La operación de adyacencia es aplicable tanto a los grafos dirigidos como a los no dirigidos, pero la enumeración de los arcos que salen de un vértice solamente es aplicable a los grafos dirigidos.

En ambos casos existe una **jerarquía**: entre el supertipo “matriz”, y los subtipos “matriz rectangular” y “matriz dispersa”, en un caso, y entre el supertipo “grafo” y los subtipos “grafo dirigido” y “grafo no dirigido”, en el otro. Esta jerarquía

se definirá en el lenguaje de programación mediante una relación de **herencia de propiedades**. Por ejemplo, la matriz dispersa es un subtipo de la matriz, por lo que todas las operaciones definidas sobre matrices (como la traspuesta) pueden aplicarse a matrices dispersas, pero la implementación de éstas es diferente según el tipo de matriz: no se programan igual las trasposiciones de una matriz dispersa y de una matriz rectangular. De la misma forma, el grafo dirigido es un subtipo del grafo, que hereda todas las operaciones definidas sobre grafos y en el que se pueden definir nuevas operaciones aplicables solamente a grafos dirigidos, como enumerar por separado los arcos que salen y entran de un nodo.

6.1.1. Subtipado en C++

El que una clase sea un subtipo de otra se indica posponiendo el nombre del supertipo al del subtipo, separado por dos puntos (:) de la forma siguiente:

```
class subtipo : acceso supertipo {
    // Declaración o definición de la clase subtipo
};
```

donde `acceso` puede ser `public`, `private` o `protected`. Esta palabra indica si los miembros públicos del supertipo son también miembros públicos del subtipo.

Los subtipos heredan los datos y funciones miembros de la clase `supertipo` que han sido declarados `public` o `protected`. En el primer caso, (`public`) la definición equivale a marcar como públicos en el subtipo los datos y funciones heredados: son accesibles desde cualquier punto del programa. En el otro caso (`protected`) son accesibles desde el subtipo pero no desde otro punto del programa; es análogo a marcar como privados los datos y funciones que se hayan heredado. Los datos y funciones marcados como `private` no se heredan.

Observe que un subtipo tiene, en general, más miembros que el supertipo, porque hereda todos los declarados como `public` y `protected` del supertipo y puede tener además miembros propios.

Por ejemplo, los ficheros de entrada (`ifstream`) y los ficheros de salida (`ofstream`) son subtipos de una clase "fichero" llamada `fstreambase`. La clase `fstreambase` no se emplea directamente en el programa; sirve para definir las operaciones comunes a todos los tipos de ficheros, de la siguiente forma:

```
class fstreambase {
   ...
   fstreambase();
   fstreambase(const char *name, int mode, int prot=0664);
   void open(const char *name, int mode, int prot=0664);
   ...
};

class ifstream : public fstreambase {
   ...
   ifstream() : fstreambase() { }
   ifstream(const char *name, int mode=ios::in, int prot=0664)
        : fstreambase(name, mode | ios::in, prot) { }
```

```
    void open(const char *name, int mode=ios::in, int prot=0664)
         { fstreambase::open(name, mode | ios::in, prot); }
    ...
};

class ofstream : public fstreambase {
    ...
    ofstream() : fstreambase() { }
    ofstream(const char *name, int mode=ios::out, int prot=0664)
         : fstreambase(name, mode | ios::out, prot) { }
    void open(const char *name, int mode=ios::out, int prot=0664)
         { fstreambase::open(name, mode | ios::out, prot); }
    ...
};
```

Obsérvese la sintaxis que se ha usado en el constructor: los dos puntos (:) tras el nombre del constructor del subtipo, seguidos por la llamada al constructor del supertipo indican que debe llamarse al constructor de `fstreambase` antes de ejecutar el código del constructor de `ifstream`, por ejemplo.

La función miembro `open` se redefine en los subtipos. En la superclase esta operación necesita un argumento que indique si el fichero se abre para leer datos o para escribir en él. En la subclase `ifstream` la orden `open` tiene el valor por defecto "lectura" y en la subclase `ofstream` tiene el valor por defecto "escritura". Obsérvese la sintaxis necesaria para que las llamadas a `open` de `ifstream` y `ofstream` se refieran a la función definida en `fstreambase` y no a una llamada recursiva al miembro `open` de `ifstream`.

6.2. Acciones virtuales y tipos polimórficos. Prog. orientada al objeto

6.2.1. Acciones virtuales

Puede darse que la unión de los subtipos coincida con el supertipo. Si una operación está definida de forma diferente en todos los subtipos, no es necesario definirla en el supertipo, porque nunca será utilizada. Basta con declarar que existe y que es aplicable al supertipo y, por consiguiente, a todos sus subtipos.

Una función miembro que va a ser redefinida en todos los subtipos se denomina **función miembro virtual** . Mediante una declaración de una función virtual indicamos que esa función está definida sobre el tipo y que su definición es dependiente de la representación del subtipo.

6.2.2. Tipos polimórficos. Programación orientada al objeto

Un tipo con funciones virtuales se llama **tipo polimórfico**. Los lenguajes de programación que soportan **herencia** y **polimorfismo** se llaman también **lenguajes orientados al objeto**.

6.2.3. Tipos polimórficos en C++

Los métodos se declaran anteponiendo la palabra `virtual` a la declaración de la función miembro correspondiente. Aunque es posible escribir código para una función virtual, es frecuente que se deje sin definir, lo que se representa con una asignación a cero.

6.2.4. Ejemplo

En el ejemplo que sigue se utilizan los conceptos de herencia de propiedades, acción virtual y tipo polimórfico. Para ello se definirán varios tipos de datos relacionados con el juego del ajedrez y se escribe un código que dibuja en la pantalla del ordenador un tablero de ajedrez con todas sus piezas en la posición inicial. Estos tipos son:

1. `pieza`, que modela al conjunto compuesto por las piezas del juego. Una pieza puede ser `peón`, `torre`, `caballo`, `alfil`, `reina` o `rey`, luego `peón` es un subtipo de `pieza`, y heredará sus propiedades.
2. `casilla`, que representa una casilla del tablero
3. `tablero`, que representa al tablero completo.

Las propiedades de estos tipos son:

1. Todas las piezas pueden ser blancas o negras. Cada subtipo de pieza tiene un símbolo que la caracteriza. Por ejemplo, un peón tiene una 'P' y una torre una 'T'.
2. Las casillas pueden contener o no una pieza. Una casilla es blanca o negra.
3. El tablero contiene 64 casillas, 32 blancas y 32 negras.

Las operaciones necesarias son:

1. Tipo `pieza`
   - La operación `muestra_simbolo` dibuja el símbolo que caracteriza a la pieza.
   - La operación `dibuja`, aplicada a una pieza, muestra su símbolo correspondiente, con caracteres resaltados si la pieza es blanca y en intensidad normal si la pieza es negra.

   Observe que la operación `muestra_simbolo` es virtual pura y está redefinida en todos los subtipos, por lo que `pieza` es un tipo polimórfico. Además, todos los subtipos heredan la definición de la operación `dibuja` de la superclase `pieza`.
2. Tipo `casilla`
   - La operación `pon_pieza` asocia una pieza con una casilla.
   - La operación `quita_pieza` deshace la asociación entre una pieza y la casilla.

- La operación `dibuja` muestra un espacio en blanco o el símbolo de una pieza, según proceda. El fondo se muestra en blanco, si la casilla es blanca, o en negro, si la casilla es negra.

3. Tipo `tablero`:

   - La operación `dibuja` aplicada al tablero muestra todas sus casillas en su lugar correspondiente en la pantalla.

```
#include <iostream>
#include <fstream>
#include <vector>
using namepsace std;

namespace vt100 {

     void cursor(int x, int y) {
        cout << (char)27 << "[" << y << ";" << x << "f";
     }

     void borra_pantalla() {
        cout << (char)27 << "[2J";
        cout << (char)27 << "[f";
     }

     void fondo_inverso() {
        cout << (char)27 << "[7m";
     }

     void fondo_normal() {
        cout << (char)27 << "[27m";
     }

     void letra_negrita() {
        cout << (char)27 << "[1m";
     }

     void letra_normal() {
        cout << (char)27 << "[0m";
     }


};

class pieza {
   public:
     typedef enum colores { blanco, negro };
   protected:
     colores color;
```

```
   public:

     // Constructores
     pieza() { }
     pieza(colores c) { color=c; }
     pieza(const pieza &p) { color=p.color; }

     // Asignación
     void operator=(const pieza &p) { color=p.color; }

     // Representación gráfica
     virtual void muestra_simbolo()=0;
     void dibuja() {
       if (color==negro)
         vt100::letra_negrita();
       muestra_simbolo();
       if (color==negro)
         vt100::letra_normal();
     }
};

class peon : public pieza {
    public:
    peon(colores c) : pieza(c) { }
    void muestra_simbolo() { cout << "P"; }
};

class torre : public pieza {
    public:
    torre(colores c) : pieza(c) { }
    void muestra_simbolo() { cout << "T"; }
};

class caballo : public pieza {
    public:
    caballo(colores c) : pieza(c) { }
    void muestra_simbolo() { cout << "C"; }
};

class alfil : public pieza {
    public:
    alfil(colores c) : pieza(c) { }
    void muestra_simbolo() { cout << "A"; }
};

class reina : public pieza {
    public:
    reina(colores c) : pieza(c) { }
```

```
    void muestra_simbolo() { cout << "Q"; }
};

class rey : public pieza {
    public:
    rey(colores c) : pieza(c) { }
    void muestra_simbolo() { cout << "K"; }
};


class casilla {
     public:
     typedef enum colores { blanco, negro };
     private:
     colores color;
     pieza *p;
     public:

     // Constructores y asignación
     casilla(colores c) { color=c; p=NULL; }
     casilla(const casilla &cs) {
        color=cs.color; p=cs.p;
     }
     void operator=(const casilla &cs) {
        color=cs.color; p=cs.p;
     }

     // Poner o quitar piezas de la casilla
     void pon_pieza(pieza *pz) { p=pz; }
     void quita_pieza() { p=NULL; }

     // Mostrar la casilla en pantalla
     void dibuja() {
         if (color==negro)
           vt100::fondo_inverso();
         if (p!=NULL) p->dibuja(); else cout << ' ';
         if (color==negro)
           vt100::fondo_normal();
     }
};


class tablero {
    vector<vector<casilla> > contenido;
    public:
    tablero(const vector<pieza*> &pz);
    void dibuja();
};
```

```
tablero::tablero(const vector<pieza*> &pz) {
     int x,y,j;

     // Creamos las casillas
     contenido=vector<vector<casilla> >(8,
         vector<casilla>(8,casilla(casilla::blanco)));

     // Coloreamos las casillas
     for (x=0;x<contenido.size();x++)
       for (y=0;y<contenido[x].size();y++)
          if ((x+y)%2==1) contenido[x][y]=casilla(casilla::negro);

     // Asociamos cada pieza a su casilla
     j=0;

     // Peones
     for (x=0;x<contenido.size();x++,j++) contenido[x][1].pon_pieza(pz[j]);
     for (x=0;x<contenido.size();x++,j++) contenido[x][6].pon_pieza(pz[j]);

     // Torres
     contenido[0][0].pon_pieza(pz[j++]);
     contenido[7][0].pon_pieza(pz[j++]);
     contenido[0][7].pon_pieza(pz[j++]);
     contenido[7][7].pon_pieza(pz[j++]);

     // Caballos
     contenido[1][0].pon_pieza(pz[j++]);
     contenido[6][0].pon_pieza(pz[j++]);
     contenido[1][7].pon_pieza(pz[j++]);
     contenido[6][7].pon_pieza(pz[j++]);

     // Alfiles
     contenido[2][0].pon_pieza(pz[j++]);
     contenido[5][0].pon_pieza(pz[j++]);
     contenido[2][7].pon_pieza(pz[j++]);
     contenido[5][7].pon_pieza(pz[j++]);

     // Reinas
     contenido[3][0].pon_pieza(pz[j++]);
     contenido[4][7].pon_pieza(pz[j++]);

     // Reyes
     contenido[4][0].pon_pieza(pz[j++]);
     contenido[3][7].pon_pieza(pz[j++]);
}
```

```
void tablero::dibuja() {
   vt100::borra_pantalla();
   for (int x=0;x<contenido.size();x++)
     for (int y=0;y<contenido[x].size();y++) {
       vt100::cursor(x+1,y+1);
       contenido[x][y].dibuja();
     }
   vt100::cursor(0,20);
}

int main() {

   int i,j=0;

   vector<pieza*> piezas=vector<pieza*>(32);
   for (i=0;i<8;i++,j++) piezas[j]=new peon(pieza::blanco);
   for (i=0;i<8;i++,j++) piezas[j]=new peon(pieza::negro);
   for (i=0;i<2;i++,j++) piezas[j]=new torre(pieza::blanco);
   for (i=0;i<2;i++,j++) piezas[j]=new torre(pieza::negro);
   for (i=0;i<2;i++,j++) piezas[j]=new caballo(pieza::blanco);
   for (i=0;i<2;i++,j++) piezas[j]=new caballo(pieza::negro);
   for (i=0;i<2;i++,j++) piezas[j]=new alfil(pieza::blanco);
   for (i=0;i<2;i++,j++) piezas[j]=new alfil(pieza::negro);
   piezas[j++]=new reina(pieza::blanco);
   piezas[j++]=new reina(pieza::negro);
   piezas[j++]=new rey(pieza::blanco);
   piezas[j++]=new rey(pieza::negro);

   tablero t(piezas);
   t.dibuja();

   for (int i=0;i<piezas.size();i++) delete piezas[i];

}
```

Índice alfabético

árbol binario, 95

abstracción de datos, 64
acción, 9, 11
algoritmo, 11
almacenamiento libre, 78
análisis, 14
array, 34
atributos, 65
automáticas, 51

bit, 13
bloque, 26
booleano, 19
byte, 13

camino en un grafo, 100
carácter, 19
ciclo, 100
codificación, 14
composición condicional, 26
composición for, 31
composición iterativa, 26
composición secuencial, 26
composición while, 30
composicion do-while, 31
constante, 17
constructor, 78
constructor de copia, 81

datos, 9, 11
datos de entrada, 54
datos de entrada y salida, 54
datos de salida, 54
datos miembro, 65
do-while, 30

encapsulación de datos, 64
entero, 17
entorno, 9
entrada, 25
escritura, 25
especificación de requisitos, 14
estado, 11

false, 19
fichero, 105
ficheros de entrada, 105
ficheros de salida, 105
for, 30
función miembro virtual, 111
funciones miembro, 65

globales, 41
grafo acíclico, 100
grafo conectado, 100

herencia, 111
herencia de propiedades, 110

identificador, 21
indicadores, 11
interface, 76

jerarquía, 109

léxico, 10, 11
lectura, 25
lenguajes orientados al objeto, 111
locales, 41
longitud de un camino, 100

módulo, 55
matriz de adyacencias, 101
matriz dispersa, 90
memoria, 12
memoria dinámica, 78

número binario, 13

nombre de una variable, 17

objeto, 17
objeto de almacenamiento, 17
ocultación de información, 64

parámetro formal, 46
parámetro real, 46
parametrizar, 45
paso por referencia, 53
paso por valor, 53
periféricos, 12
polimorfismo, 111
primitiva, 10
private, 65
procedimientos, 44
procesador, 9
programas, 10
protected, 65
prototipos, 47
public, 65
puntero, 50

real, 18
recorrido, 35
recorrido en anchura, 101
recorrido en profundidad, 100
referencia, 50
refinamientos sucesivos, 41
resultado, 11

salida, 25
sinónimos, 50
sobrecarga, 48
static, 51
subgrafo, 100
subrutinas, 44
subtipo, 109
supertipo, 109

this, 67
tipo, 12
Tipo Abstracto de Datos, 64
tipo base, 34
tipo de datos, 17
tipo polimórfico, 111
tipos paramétricos, 83
true, 19

unidad central de proceso, 12

vértices adyacentes, 100
variable, 17
vector, 34
verificación, 14

while, 30