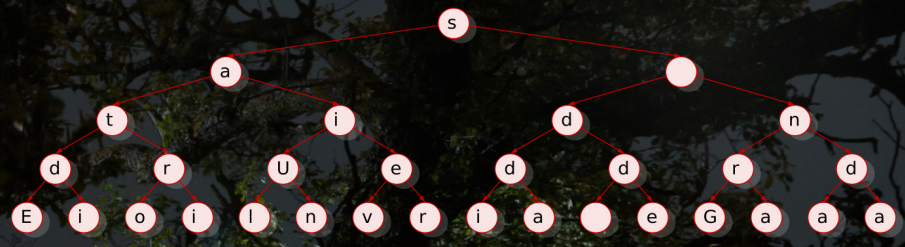
Estructuras de datos avanzadas

con soluciones en C++

Antonio Garrido Carrillo



eug

ESTRUCTURAS DE DATOS AVANZADAS: con soluciones en C++
ISBN: 978-84-338-6409-3
Edita: Editorial Universidad de Granada
Campus Universitario de Cartuja. Granada.
Edición digital.

.

Índice general

1 Vectores y celdas enlazadas 1

1.1 Introducción 1

1.2 Vectores en memoria dinámica 1

1.2.1 Vectores de tamaño cero y punteros nulos 2
1.2.2 Ejemplos de reservar/liberar estructuras de vectores 3
1.2.3 Almacenamiento automático vs dinámico 9
1.2.4 Rangos en un vector 12

1.3 Celdas enlazadas 13

1.3.1 Recorrer una lista 14
1.3.2 Insertar y borrar una celda 15
1.3.3 Celda controlada desde la anterior 17
1.3.4 Rango en una lista de celdas 18

1.4 Vectores vs celdas enlazadas 19

1.5 Problemas 20

2 Análisis avanzado de eficiencia 23

2.1 Introducción 23

2.2 Análisis de algoritmos recursivos 24

2.2.1 Ecuación recurrente 24
2.2.2 Ecuación recurrente simplificada 25

2.3 Deducción de la expresión general 26

2.3.1 Cambio de variable 28

2.4 Ecuación característica 30

2.4.1 Recurrencias Homogéneas Lineales 31
2.4.2 Recurrencias no Homogéneas 33
2.4.3 Ecuaciones recurrentes no lineales: cambios de variable 35

2.5 Análisis amortizado 40

2.5.1 Calculando por fuerza bruta 41
2.5.2 Método contable 45

2.5.3 Funciones de potencial . . . 48
2.5.4 Mezcla de operaciones . . . 52
2.6 Problemas 57

3 Pilas y colas . . . 59
3.1 Introducción 59
3.2 La clase *Pila* 60
3.2.1 Implementaciones . . . 62
3.2.2 Implementación con vector de tamaño fijo . . . 63
3.2.3 Implementación con vector de tamaño variable . . . 64
3.2.4 Implementación basada en celdas enlazadas . . . 66
3.3 La clase *Cola* 70
3.3.1 Implementaciones . . . 72
3.3.2 Implementación basada en vectores . . . 72
3.3.3 Implementación basada en celdas enlazadas . . . 78
3.4 Modificaciones sobre tipos *Pila* y *Cola* 81
3.4.1 Plantillas . . . 81
3.4.2 Mejoras en la interfaz . . . 82
3.4.3 Mejoras en la implementación . . . 83
3.4.4 Colas limitadas . . . 86
3.5 La clase *Cola doble* 88
3.5.1 Implementaciones . . . 89
3.5.2 Implementación basada en celdas enlazadas . . . 90
3.5.3 Implementación con vectores por bloques . . . 92
3.6 La clase *Cola con prioridad* 100
3.6.1 Implementación . . . 103
3.6.2 Orden de prioridad como parámetro de plantilla . . . 105
3.7 Problemas 106

4 Listas . . . 109
4.1 Introducción 109
4.2 La clase *Lista* 110
4.2.1 Posiciones en una lista . . . 110
4.2.2 Interfaz propuesta . . . 111
4.3 Implementación del tipo *Lista* 117
4.3.1 Representaciones de los tipos . . . 119
4.3.2 Implementación de iteradores . . . 120
4.3.3 Inserción y borrado . . . 121
4.4 La clase *ListaSimple* 122
4.5 Implementación de *ListaSimple* 123
4.5.1 Implementación con celdas simplemente enlazadas . . . 123
4.5.2 Implementación con celdas simplemente enlazadas y cabecera . . . 128

4.6 La clase *ListaFwd* 132
4.6.1 Inserción y borrado . . . 133
4.7 Implementación de *ListaFwd* 135
4.7.1 Inserción y borrado . . . 136
4.8 Detalles avanzados de implementación 136
4.8.1 Constructores para el tipo *Celda* . . . 137
4.8.2 Construcción y asignación por movimiento . . . 138
4.8.3 Representación de la cabecera . . . 140
4.8.4 Gestión de excepciones . . . 142
4.8.5 Constructor por movimiento y *noexcept* . . . 143
4.9 Iteradores y programación genérica 145
4.9.1 Abstracción por iteración . . . 145
4.9.2 Tipos de iteradores . . . 146
4.10 Problemas 148

5 Árboles generales . . . 151

5.1 Introducción y terminología básica 151
5.1.1 Definiciones básicas . . . 152
5.1.2 Recorridos . . . 155
5.2 Representación para árbol general 160
5.2.1 Representación con celdas enlazadas . . . 161
5.2.2 Moviéndose por los nodos del árbol . . . 162
5.2.3 Creación y modificación de árboles . . . 163
5.2.4 Liberar un árbol completo . . . 165
5.2.5 Listar elementos de un árbol general . . . 167
5.3 La clase *ArbolGen* 170
5.3.1 Interfaz pública del tipo *ArbolGen* . . . 171
5.3.2 Implementación del tipo *ArbolGen* . . . 175
5.3.3 Iteradores sobre un árbol general . . . 178
5.4 E/S de árboles. Serialización 182
5.4.1 Árbol desde recorridos . . . 182
5.4.2 Preorden más profundidad . . . 183
5.4.3 Preorden con marcas o centinelas . . . 184
5.4.4 Preorden/postorden: principio y fin . . . 185
5.5 Problemas 187

6 Árboles n-arios . . . 189

6.1 Introducción y terminología básica 189
6.1.1 Definiciones básicas . . . 190
6.2 Representación de árboles n-arios 192
6.2.1 Representación en un *vector de etiquetas* . . . 192
6.2.2 Representación con *celdas enlazadas/vector de hijos* . . . 194

6.3 La clase NArbol **195**
6.3.1 Interfaz pública del tipo *NArbol* . . . 195
6.3.2 Implementación del tipo *NArbol* . . . 200
6.3.3 Iteradores sobre un *NArbol* . . . 203
6.4 B-árboles **205**
6.4.1 Introducción y terminología básica . . . 205
6.4.2 Representación de *B-árboles* . . . 209
6.4.3 La clase *BArbol* . . . 211
6.4.4 *B-árboles* y almacenamiento externo . . . 224
6.4.5 Árboles *B+* y *B** . . . 227
6.5 Problemas **230**

7 Árboles binarios . . . 233
7.1 Introducción y terminología básica **233**
7.1.1 Recorridos . . . 234
7.1.2 Ejemplo: árboles de expresión . . . 235
7.2 Representación de árboles binarios **237**
7.2.1 Moviéndose por los nodos del árbol . . . 238
7.2.2 Creación y modificación de árboles . . . 239
7.2.3 Liberar un árbol . . . 240
7.2.4 Listar elementos de un árbol binario . . . 241
7.3 La clase *ArbolBin* **243**
7.3.1 Interfaz pública del tipo *ArbolBin* . . . 244
7.3.2 Iteradores sobre un árbol binario . . . 248
7.3.3 Implementación del tipo *ArbolBin* . . . 249
7.4 E/S de árboles binarios. Serialización **254**
7.4.1 Árbol binario desde recorridos . . . 254
7.4.2 Preorden con marcas . . . 255
7.5 Problemas **255**

8 Árboles binarios de búsqueda . . . 257
8.1 Introducción **257**
8.2 ABB simple **258**
8.2.1 Búsqueda en un ABB . . . 258
8.2.2 Inserción . . . 259
8.2.3 Borrado . . . 261
8.2.4 Eficiencia . . . 263
8.2.5 Implementación en base a árbol binario . . . 265
8.3 Árboles desplegados **267**
8.3.1 Buscar... ¡modifica el árbol! . . . 268
8.3.2 Despliegue . . . 270
8.3.3 Unión y división . . . 273

8.3.4 Búsquedas, inserciones y borrados . . . 274
8.3.5 Eficiencia . . . 278

8.4 Árboles AVL 281
8.4.1 Definición . . . 281
8.4.2 Inserción . . . 283
8.4.3 Borrado . . . 287
8.4.4 Eficiencia . . . 289
8.4.5 Implementación básica con punteros . . . 292
8.4.6 Implementación en base a árbol binario . . . 296

8.5 Árboles Rojo-Negro 298
8.5.1 Nodos externos negros . . . 300
8.5.2 Búsqueda, inserción y borrado . . . 302
8.5.3 Eficiencia de un árbol rojo-negro . . . 321
8.5.4 Relación con árboles 2-4 . . . 323

8.6 Problemas 325

9 Colas con prioridad: heaps . . . 329

9.1 Introducción y terminología básica 329

9.2 Heaps: operaciones básicas 330
9.2.1 Operaciones: *frente, poner, quitar* . . . 331
9.2.2 Implementación sobre un vector-C . . . 332
9.2.3 Cola con prioridad sobre un *heap* . . . 336
9.2.4 Heaps k-arios . . . 338

9.3 Heaps oblicuos: Mezcla 339
9.3.1 Heaps oblicuos . . . 340
9.3.2 Implementación . . . 342
9.3.3 Eficiencia . . . 345
9.3.4 Mezcla *top-down vs bottom-up* . . . 349

9.4 Heaps a izquierda 349
9.4.1 Equilibrio de *heap a izquierda* . . . 349
9.4.2 Camino derecho y número de nodos . . . 351
9.4.3 Mezclar en un *heap a izquierda* . . . 353
9.4.4 Implementación . . . 353

9.5 Decrementar 355
9.5.1 Decrementar en *heap a izquierda* . . . 357
9.5.2 Implementación . . . 359
9.5.3 Borrado de un nodo cualquiera . . . 361

9.6 Problemas 362

10 Bosques . . . 363

10.1 Introducción 363

10.2 Heaps binomiales **364**
10.2.1 Árbol binomial . . . 364
10.2.2 Definición . . . 367
10.2.3 Operaciones . . . 368
10.2.4 Eficiencia . . . 371
10.2.5 Implementación . . . 372
10.3 Heaps binomiales perezosos **377**
10.3.1 Operaciones perezosas . . . 377
10.3.2 El tipo *Bosque* . . . 378
10.3.3 Mezcla perezosa . . . 382
10.3.4 Implementación . . . 384
10.4 Heaps Fibonacci **390**
10.4.1 Árboles Fibonacci . . . 390
10.4.2 Tamaño y número de hijos de un nodo . . . 394
10.4.3 Análisis amortizado . . . 397
10.4.4 Implementación . . . 398
10.4.5 Borrado de un nodo cualquiera . . . 403
10.5 Conjuntos disjuntos **405**
10.5.1 Relaciones de equivalencia: ejemplos . . . 405
10.5.2 El bosque: un árbol por cada conjunto . . . 407
10.5.3 Unir/buscar . . . 408
10.5.4 Optimización . . . 410
10.5.5 Implementación . . . 413
10.6 Problemas **416**

11 Tablas Hash . . . 419

11.1 Introducción **419**
11.1.1 Un ejemplo muy simple . . . 420
11.2 Funciones hash **420**
11.2.1 Diseño de funciones hash . . . 422
11.2.2 Hashing de un entero . . . 423
11.2.3 Hashing de una cadena . . . 425
11.2.4 Usos de funciones *hash* . . . 427
11.2.5 Hashing en el estándar de C++ . . . 428
11.3 Resolución de Colisiones **430**
11.3.1 Hashing cerrado o direccionamiento abierto . . . 431
11.3.2 Hashing abierto o encadenamiento separado . . . 437
11.3.3 Encadenamiento mezclado . . . 438
11.4 Eficiencia de las tablas hash **441**
11.4.1 Factor de carga . . . 442
11.4.2 Comparación de métodos . . . 442
11.4.3 Redimensionamiento y rehashing . . . 445
11.4.4 Tablas hash vs árboles de búsqueda . . . 446

11.5 Implementación **447**
11.5.1 Hashing con cursores 447
11.5.2 Hashing con iteradores 455
11.6 Problemas. **464**

A Solución a los ejercicios **467**
A.1 Vectores y celdas enlazadas **467**
A.2 Eficiencia **471**
A.3 Pilas y colas **477**
A.4 Listas **485**
A.5 Árboles generales **491**
A.6 Árboles n-arios **497**
A.7 Árboles binarios **505**
A.8 Árboles binarios de búsqueda **508**
A.9 Colas con prioridad: heaps **518**
A.10 Bosques **522**
A.11 Tablas Hash **528**

B Tablas **535**
B.1 Tabla ASCII **535**
B.2 Operadores C++ **535**
B.3 Palabras reservadas de C89, C99, C11, C++ y C++11 **538**

Bibliografía **541**
Referencias electrónicas **544**

Índice alfabético **545**

Prólogo

El objetivo de este libro es presentar un curso de estructuras de datos en C++ para un estudiante que tiene conocimientos básicos sobre encapsulamiento en este lenguaje pero que desconoce la mayoría de las estructuras de datos. Por tanto, el curso abarca contenidos que van desde estructuras simples como las pilas y colas hasta estructuras más avanzadas como árboles equilibrados, estructuras autoajustables o tablas hash.

En primer lugar, un curso de estructuras de datos no tiene por qué situarse en el contexto del lenguaje C++. Se podría presentar incluso en lenguaje algorítmico, evitando detalles incómodos de implementación; centrándose en los conceptos, las estructuras de datos, los algoritmos y la eficiencia. En este sentido, podemos decir que las estructuras de datos se pueden estudiar a distintos niveles; como consecuencia, podrá encontrar distintos enfoques en la bibliografía, desde cursos básicos sobre cómo estructurar la información hasta detalles sobre cómo implementarlo en algún lenguaje concreto. Existen muchas referencias en otros lenguajes o incluso en varios lenguajes.

Por ejemplo, un curso basado en el uso —evitando detalles— de las estructuras de datos básicas que maneja la STL se puede encontrar en Garrido[2], donde se estudia la programación genérica en C++ y para la que es conveniente entender en qué consisten las estructuras de datos que sustentan los contenedores de la biblioteca estándar. En este caso, es fundamental entender cómo está estructurada la información para poder manejar con la máxima eficacia cada uno de los contenedores.

Sin embargo, para asimilar en profundidad estos conocimientos sobre cómo funcionan estas estructuras de datos, es conveniente que el estudiante se enfrente a los detalles de implementación. Además, para un estudiante que no tiene demasiada experiencia programando, resulta de especial interés poder practicar en un lenguaje concreto, lo que le hará asimilar mucho mejor el funcionamiento de estas estructuras; no sólo eso, además, le hará comprender cómo se pueden resolver los detalles que afectan a una buena implementación no sólo de estas estructuras, sino en cualquier algoritmo en general. Por ejemplo, podrá practicar y mejorar sus conocimientos sobre abstracción y diseño modular.

Por tanto, en este libro se van a abordar detalles de implementación con C++. El objetivo es que el estudiante conozca muy bien cómo puede crearse el código que resuelve el problema de la estructura de datos de forma que se garantice que las conoce en su totalidad, sin dudas o ambigüedades. Además, practicará con problemas que plantean distintas soluciones y para los que conviene reconocer la mejor alternativa, tanto en términos de eficiencia como de calidad del código obtenido.

Tenga en cuenta que implementar una estructura de datos obliga a reflexionar sobre todos y cada uno de los detalles. Puede estar convencido de conocer al 100 % una estructura o algoritmo y encontrarse con dudas cuando quiere incluir los detalles en un programa. Por ejemplo, durante la implementación de los tipos y algoritmos que se presentan en este libro, algunas veces me ha

resultado esclarecedor tener que resolver sutilizas que sin la implementación no hubiera descubierto. Por supuesto, detalles que de una u otra forma he procurado que aparezcan en este texto.

Por otro lado, no debe considerarse un curso avanzado de programación en C++. De hecho, en algunos casos se esquivarán detalles que podrían oscurecer la discusión. Se evitarán en gran medida diseños basados en programación dirigida a objetos y metaprogramación, sin olvidar que en algún caso se incluirán algunas aportaciones que se añaden, inevitablemente, para evitar una mala solución.

En lugar de ello, se facilitará la descripción de los algoritmos mediante una implementación que especifica con precisión los detalles sobre la estructura de datos, sin intención de obtener versiones especialmente optimizadas para producción. Como bien decía Albert Einstein: *hazlo tan simple como sea posible, pero no más*.

A pesar de todo, en algún caso se introducirán algunos detalles avanzados de implementación que ayudarán a entender mejor el lenguaje y sus ventajas. En este sentido, gran parte de las implementaciones se podrían realizar en C++98, pero se irán introduciendo algunos contenidos de C++11 o posteriores. La idea es que no sea necesario tener experiencia en los estándares actuales y, para aquellos estudiantes que aún no la tengan, pueda servir para mejorar su formación en ellos.

No olvidemos que, si bien no es nuestra intención centrarnos en la mejor implementación, si debemos presentar buenas soluciones. Además, los nuevos estándares, si bien son más complejos al ser mucho más amplios, también incluyen utilidades que facilitan la implementación. En parte, uno de los efectos positivos de estudiar este texto es que el lector debería sentirse más cómodo usando algunas de las nuevas características de C++11 y posteriores.

Qué y para quién es este libro

Este libro tiene su génesis en un proyecto docente personal que intenta aportar un material de trabajo especialmente diseñado para los estudiantes de *Ingeniería Informática* de la *Universidad de Granada*. El origen está en un contexto universitario, pero el desarrollo se ha planteado de una forma genérica pensando en qué debería conocer un buen programador.

La primera opción es identificar contenidos con las asignaturas que componen un plan de estudios. Sin embargo, es bien sabido que la forma en que se diseñan no es precisamente pensando en que funcionen gracias a una fuerte interrelación. Normalmente, se intentan hacer compartimentos estancos que simplifiquen el diseño de contenidos y la docencia.

Son muchas las cosas y muchas más las relaciones entre ellas para poder formular una solución simple en forma de secuencia de contenidos que aprender. No es difícil identificar los sillares que podrían cimentar la formación de un programador, si bien es bastante complejo adivinar la mejor forma de imbricarlos para obtener la base ideal.

El resultado del proyecto es un conjunto de libros que apoyándose en el lenguaje C++ —*"a modo de mortero"*— intentan ir componiendo los fundamentos para garantizar una buena base sobre los que avanzar en su formación. Es importante que no queden huecos para poder cerrar etapas y abrir nuevos retos con la mejor garantía.

El libro que nos ocupa constituye un verdadero sillar ciclópeo en este proyecto. Por un lado, necesita que el lector tenga una formación básica bien asentada. Las referencias Garrido[5][1] y en parte Garrido[2] contienen fundamentos más que suficientes para poder enfrentarse a este libro. Por otro, incluye contenidos que amplían los conocimientos en varios aspectos, pudiendo ser útiles en asignaturas de estructuras de datos, pero también en temas relacionados con algorítmica o bases de datos; además, no sólo aporta en temas fundamentales, sino que incluye temas avanzados con estructuras de datos complejas que normalmente no aparecen en cursos básicos de programación.

En conclusión, el libro no es tanto el texto de una materia concreta, sino un nuevo bloque de contenidos para un estudiante interesado en la programación. Puede usarse para estudios

universitarios, pero también puede servir para estudios medios, autodidactas e incluso algunos temas en estudios más avanzados.

Para quien no es el libro es para el programador ocasional. Ese programador que necesita rápidamente conocer un lenguaje que le permita ejecutar y obtener unos resultados puntuales sin interés en la eficiencia o la productividad a largo plazo. Si está interesando simplemente en que "funcione" un programa, sin preocuparse en si está bien hecho o si es eficiente, este no es su libro.

Finalmente, es importante resaltar la relación con otros libros del mismo autor y que aparecen en la bibliografía. En especial, resulta muy eficaz estudiar este libro junto con Garrido[2]. Son dos referencias muy relacionadas que se alimentan una de la otra.

Tal vez, la primera parte de ese libro sea una buena base para comenzar con éste; los tres primeros temas estudian en profundidad aspectos sobre C++ que facilitan comprender los códigos de este libro. Por otro lado, estudiar los detalles más básicos de las estructuras de datos le hará comprender el porqué de muchos contenidos relacionadas con la biblioteca estándar de C++.

Organización del libro

Los capítulos se han organizado para que el lector comience desde un nivel de programación con diseño de soluciones que incluyen estructuras de datos básicas. Se supone que ya puede resolver problemas simples que, posiblemente sin soluciones óptimas, incluyen conocimientos tales como memoria dinámica, abstracción funcional, modularización, generalización y encapsulamiento.

En el **capítulo 1** se presentan fundamentos necesarios para el resto del libro. Por experiencia, los contenidos relacionadas con memoria dinámica suelen ser un problema para los estudiantes. Si son alumnos sin experiencia, no suelen haber practicado mucho o incluso si los han estudiado, les han resultado complejos. Son temas relativamente sencillos, introductorios, que se presentan de manera sobrada en otras referencias, como en Garrido[1].

A pesar de ello, son contenidos que incluyen conocimientos sobre estructuras de datos. Tal vez a bajo nivel, con dificultades limitadas pero que son una base importante que debe estar resuelta para poder avanzar en los siguientes temas. No es sólo conveniente, sino recomendable, que el estudiante revise, estudie y confirme que estos conocimientos los tiene superados.

En el **capítulo 2** se presenta un tema avanzado. No es un tema con mucho código, pues no se plantean nuevos diseños e implementaciones. Es un tema dedicado a estudiar en análisis de eficiencia. Primero se estudia el análisis de algoritmos recursivos y la resolución de recurrencias. En la segunda parte, el análisis amortizado.

Se supone que el lector ya tiene algún conocimiento básico sobre la eficiencia de algoritmos. Tenga en cuenta que los primeros programas, aun siendo simples e incluyendo estructuras de datos básicas como vectores, ya tienen una componente sobre eficiencia que se debería conocer. Por ejemplo, estudiar problemas como la búsqueda lineal o binaria requiere comentarios sobre ella.

En este sentido, podrían estudiarse las primeras estructuras de datos sin necesidad de conocer más sobre eficiencia. En realidad, el tema se podría estudiar más adelante. A pesar de ello, las estructuras de datos no pueden compararse si no se habla en términos de eficiencia. Es muy importante poder usarlos con soltura. Si a eso añadimos que nos encontramos con algoritmos recursivos y que incluso las estructuras tipo árbol tienen implementaciones directas recursivas, es absolutamente necesario que el estudiante se sienta seguro en este contexto.

En el **capítulo 3** ya introducimos los primeros tipos de datos en los que se presentan interfaces y comparaciones sobre estructuras de datos. Los primeros tipos, `Pila` y `Cola` nos van a permitir trabajar con clases, recordar conceptos básicos sobre ellas. No sólo se presentan los detalles de las estructuras sino que también se aprovecha para recordar cómo se usan en C++. En cierto sentido, es un tema sobre estructuras de datos básicas pero que incluye discusiones que permiten engarzarlo con

un curso anterior sobre metodología de programación y encapsulamiento. Si ha tenido problemas antes o necesita refrescar conocimientos sobre clases, revise detenidamente este tema.

La segunda parte presenta una estructura de datos algo más compleja: el tipo `ColaDoble`. Es bastante más laborioso que los anteriores, pero es importante conocerlo pues no deja de ser un tipo cola, aunque con una interfaz más amplia. Se podría presentar una implementación simple, por ejemplo con celdas enlazadas, pero también se incluye una discusión basada en vectores de bloques. Esta representación, más engorrosa, se estudia especialmente porque puede ser una aproximación a la solución que incluye el contenedor **`std::deque`** de la biblioteca estándar.

Finalmente, también se incluye la idea de cola con prioridad. En este caso, casi exclusivamente desde el punto de vista abstracto. Es conveniente conocer este tipo de cola aunque no tenga un comportamiento *FIFO* como las anteriores. Es una primera aproximación al problema para entender qué restricciones impone y la dificultad para conseguir una buena eficiencia. Un primer acercamiento al problema de implementar *heaps* que se estudiará en profundidad en varios capítulos más adelante.

En el **capítulo 4** seguimos con estructuras de datos lineales. El concepto sigue siendo una secuencia de elementos, pero la variedad de operaciones implica una mayor dificultad para conseguir una buena interfaz. Se discute intensivamente sobre problemas de celdas enlazadas, simples y dobles y soluciones para conseguir tipos de datos eficientes.

En este contexto, cuando la estructura no es especialmente compleja, se introducen detalles sobre iteración. Se implementan los iteradores sobre listas, revisando el concepto de **`iterator`** de C++ y viendo cómo se puede aprovechar para obtener una buena interfaz que permite la programación genérica.

Por otro lado, también se revisan otros contenidos relacionados con la implementación de clases en C++. Especialmente relacionados con las novedades de C++11, permiten afinar las implementaciones de listas creando tipos más optimizados y más cercanos a los tipos que incluye la biblioteca estándar. Además, se muestran como ejemplos tipos basados en listas enlazadas simples y dobles para que el estudiante entienda mejor las alternativas que ofrece esta biblioteca.

En el **capítulo 5** salimos de las estructuras de datos no lineales e introducimos los árboles. Las estructuras jerárquicas son un tema muy amplio, por lo que en este tema nos limitamos a hablar de los conceptos más importantes y presentar un tipo de dato *árbol general*. Se incluye un tipo de dato para manejar un árbol general que no existe en la biblioteca estándar. Está diseñado para encapsular detalles sobre implementación y discutir a un nivel de abstracción superior. Con él, no sólo se estudia otro caso de implementación en C++, sino que se comienzan a desarrollar programas basados en esa interfaz. Este tipo de implementaciones serán muy frecuentes en temas posteriores.

En el **capítulo 6** volvemos sobre las estructuras de tipo árbol pero centrándonos en *árboles n-arios*. Son árboles especialmente diseñados para tener hasta *n* hijos en posiciones concretas. Esta nueva estructura viene acompañada de nuevos conceptos y de un nuevo tipo de dato para manejar *árboles n-arios*. La última parte se dedica a estudiar en profundidad los *B-árboles*, estructuras especialmente diseñadas para la búsqueda eficiente y que resultan soluciones muy eficientes para el almacenamiento externo. Esta parte podría ser parte de un curso de bases de datos, al menos en lo que se refiere a almacenamiento físico e índices. Además, también se presentan las variantes *árbol B+* y *árbol B**.

En el **capítulo 7** nos centramos en una estructura de tipo árbol muy concreta: el árbol binario. Podríamos decir que la más sencilla, pero con una amplia oferta de aplicaciones. En este tema, nos centramos en los conceptos básicos y en proponer un nuevo tipo de dato que simplifique el uso de árboles binarios evitando los detalles a bajo nivel.

Desde el punto de vista de la estructura, no es más compleja que las anteriores. Sin embargo, es un nuevo tipo de dato abstracto, con una nueva interfaz y que aprovecharemos para introducir algunos detalles avanzados de implementación que lo hagan más útil en los temas posteriores.

En el **capítulo 8** seguimos con los árboles binarios pero en el contexto de árbol de búsqueda. Es un tema importante, como demuestra su amplitud, que incluye varias estructuras de datos. Comienza con las estructuras más simples —*ABB simple* que no garantizan eficiencia en peor caso, aunque sí en tiempo medio; sigue con estructuras autoajustables que garantizan tiempos amortizados —*árboles desplegados*— y termina con estructuras equilibradas, primero con los clásicos *árboles AVL* y terminando con los avanzados *árboles rojo-negro*.

En el **capítulo 9** presentamos las colas con prioridad. Si bien se presentaron por primera vez en los primeros capítulos, relacionándolas con las colas em FIFO, es este tema es donde se amplía y profundiza sobre este importante tipo de dato. Se comienza con la estructura de datos más usada y simple para resolver el problema de un *heap*: un vector con el que representamos un *árbol parcialmente ordenado completo*.

En la segunda parte se presentan dos estructuras de datos basadas en árboles binarios. La primera —los *heaps oblicuos*— una estructura autoajustable que garantiza tiempos amortizados, y la segunda —los *heaps a izquierda*— que garantiza tiempos en peor caso.

En el **capítulo 10** complicamos de alguna forma la estructura. En lugar de trabajar con un árbol, trabajamos con varios. Se presentan los bosques como soluciones para estructuras.

La mayor parte del tema, al principio, incluye nuevas representaciones para *heaps*. La primera son los *heaps binomiales* que garantizan tiempos logarítmicos en peor caso pero que mejoran la inserción hasta tiempo amortizado constante. A continuación se modifica mediante una opción de mezcla perezosa, lo que permite obtener también un tiempo constante en la operación de mezcla.

Después se presenta un tipo avanzado, el *heap Fibonacci*. Tiene mucho en común con los anteriores, pero complicando la estructura a fin de obtener una operación de decremento en tiempo constante. Precisamente por ello se presenta un nuevo tipo de dato: el tipo `Bosque` que permite crear un módulo como base para resolver tanto los *heaps binomiales perezosos* como los *heap Fibonacci*.

La última parte del tema se dedica a desarrollar un tipo que implemente *conjuntos disjuntos*. No está relacionado con *heaps*, pero es otra estructura basada en un bosque. La implementación, de hecho, es muy distante; se basa en un bosque representado en un único vector. Es una estructura de datos avanzada, ampliamente usada por su simplicidad y eficacia. En la práctica, sus operaciones las podemos considerar de eficiencia constante en tiempo amortizado.

En el **capítulo 11**, después de 6 dedicados a los árboles, presentamos una nueva estructura de datos, totalmente distinta a las anteriores: la *tabla hash*. Es un tema amplio que incluye nuevos conceptos y que permite entender mejor cómo funcionan los tipos `unordered` que se incorporan al estándar C++11.

En las implementaciones finales, primero se trabaja con una solución sencilla en la que se practica con cursores para manejar tablas y relaciones entre los elementos incluidos y en segundo lugar se presenta una implementación más avanzada. Esta última, bastante elaborada, pues presenta una solución con bastantes detalles de bajo nivel con la intención de cubrir esa curiosidad natural del estudiante, al que le resulta difícil entender cómo se puede pasar de esos conceptos abstractos a la interfaz que ofrece la biblioteca estándar de C++.

Además, se incluyen dos apéndices que completan el libro:

- En el **Apéndice A** se incluyen las soluciones de los ejercicios propuestos a lo largo de los temas. Es recomendable que se estudie cada tema y se revisen estos ejercicios, primero intentando solucionarlos y segundo, independientemente del resultado, consultando la solución que se propone. Los problemas que aparecen al final de los temas son propuestos para el estudiante interesando en practicar, sin soluciones en el texto.
- En el **Apéndice B** se incluyen tablas y referencias. Encontrará algunas tablas básicas, como el código *ASCII*, los operadores de C++ y la lista de palabras reservadas del lenguaje.

Finalmente, se presenta la bibliografía, donde aparece la lista de referencias que se han usado para la creación de este libro o que resultan de especial interés para que el lector amplíe conocimientos. Además, se incluye un índice alfabético que facilita al lector localizar los conceptos más relevantes que incluye este libro.

Sobre la ortografía

El libro se ha escrito cuidando la redacción y la ortografía; no podría ser de otra forma. Se ha intentado cuidar especialmente la expresión y la claridad en las explicaciones. Como parte de este cuidado, es necesario incluir en este prólogo una disculpa por no respetar todas y cada una de las normas que actualmente establece la Real Academia Española (RAE) sobre el español.

En primer lugar, es difícil concretar un lenguaje claro y duradero en el contexto de la informática, especialmente porque cambia e incorpora nuevos conceptos continuamente. Unido al retraso que requiere el análisis y la decisión por parte de la RAE para incluirlos en el diccionario de la lengua española, siempre podemos encontrar alguna palabra que se adapta desde el inglés y no está aún aceptada. Por ejemplo, usaremos la palabra *unario* como traducción de *unary*. Otros palabras que usamos de forma natural cuando comentamos código en C++ pueden ser *booleano, token, buffer, precondición, argumento, preincremento, prompt, iterador, particionar, referenciar, serializar,* etc.

Sirva como disculpa recordar cuando se escribían los textos técnicos con la palabra *iniciar* cuando queríamos decir *inicializar*: *"se tiene que iniciar la variable con cierto valor"*. Éste es un verbo que se ha incorporado más tarde al diccionario de la RAE, pero que ya se conocía y usaba intensivamente entre los programadores. Era extremadamente incómodo intentar usar su versión aceptada para el lenguaje, pues *iniciar* tiene un sentido muy distinto de lo que significa *inicializar*.

Por otro lado, es habitual que el programador trabaje con documentación técnica que está escrita en inglés. Esto hace que un uso de palabras similares en español sea una forma de facilitar la lectura rápida y cómoda. Por ejemplo, la palabra *indentación* en lugar de *sangrado*, *casting* en lugar de *moldeado*, *array* en lugar de *arreglo*, *heap* en lugar de montón o *swap* en lugar de intercambio.

Sin duda, mis cambios pueden verse como un error; pero desde un punto de vista técnico prefiero no darle excesiva importancia. Es más importante superar la dificultad de los contenidos técnicos, dejando cuestiones de la lengua en un segundo plano.

A pesar de todo, he querido ajustarme en lo posible a lo que nos dice la RAE; por respeto al estupendo y complejo trabajo que realizan, pero sobre todo por respecto a nuestra lengua materna, el principal legado común que nuestros ancestros nos han dejado.

Una vez mostrado mi respeto por nuestra lengua, tengo que disculparme por introducir alguna palabra u ortografía que no es actual. Esta disculpa se refiere, especialmente, al uso de la tilde diacrítica en el adverbio *sólo* —note la tilde— y los *pronombres demostrativos*. En mi humilde opinión, la distinción gráfica con respecto al adjetivo *solo* y los *determinantes demostrativos*, respectivamente, justifican claramente su uso.

Si quiero priorizar la legibilidad, la claridad, el énfasis de lo que comunico, creo que su uso está justificado. En este sentido, tenga que reconocer que no estoy en absoluto de acuerdo con la RAE. La lengua escrita no está diseñada para que suene igual que la hablada. En mi humilde opinión, se ha creado y moldeado con todos los detalles necesarios para hacerla precisa, evitando ambigüedades y facilitando la comprensión del lector.

Por supuesto, seguramente habrá otros errores ortográficos o gramaticales; éstos son fruto de mi torpeza e ignorancia, por los que pido disculpas.

Agradecimientos

En primer lugar, quiero agradecer el trabajo desinteresado de miles de personas que han contribuido al desarrollo del software libre, sobre el que se ha desarrollado este material, y que

constituye una solución ideal para que el lector pueda disponer del software necesario para llevar a cabo este curso.

Por otro lado, no puedo olvidar a las personas que me han animado a seguir trabajando para crear este documento. El trabajo y tiempo que implica escribir un libro no viene compensado fácilmente si no es por las personas que justifican ese esfuerzo. En este sentido, quiero agradecer a mis estudiantes su paciencia, y recordar especialmente a aquellos, que con su esfuerzo e interés por aprender, han sabido entender el trabajo realizado para facilitar su aprendizaje.

Tengo que confesar que son ellos, los alumnos que desean aprender, los que llegan con una inmensa ilusión por recibir los conocimientos de un supuesto experto en un tema, los que me animan a seguir trabajando en este tipo de material; especialmente si tenemos en cuenta que no sólo no se premia, sino que prácticamente no se reconoce este trabajo. Sirva como una humilde aportación a lo que me hubiera gustado encontrar cuando llegué a la universidad.

A. Garrido.
Septiembre de 2018.

1 Vectores y celdas enlazadas

Introducción 1
Vectores en memoria dinámica 1
- Vectores de tamaño cero y punteros nulos
- Ejemplos de reservar/liberar estructuras de vectores
- Almacenamiento automático vs dinámico
- Rangos en un vector

Celdas enlazadas 13
- Recorrer una lista
- Insertar y borrar una celda
- Celda controlada desde la anterior
- Rango en una lista de celdas

Vectores vs celdas enlazadas 19
Problemas 20

1.1 Introducción

Estudiar estructuras de datos en el contexto de C++ requiere conocer con cierto detalle las herramientas básicas que el lenguaje nos ofrece para crear estructuras en memoria dinámica.

Este libro está escrito considerando que el lector ya lo ha estudiado en cursos anteriores. En concreto, los contenidos de las referencias Garrido[1][5] pueden ser una buena base para garantizar estos fundamentos.

A pesar de ello, es importante revisar algunos de estos conocimientos básicos para hacer el libro más "autocontenido", al menos, en aquellos aspectos menos concretos que, por experiencia, demuestran ser un problema para el estudiante.

Por tanto, para comenzar con las estructuras de datos más básicas en este curso, es necesario recordar y afianzar los detalles sobre cómo se manejan estructuras en memoria dinámica.

En este capítulo vamos a revisar algunos de estos conocimientos antes de entrar en la implementación de las distintas estructuras de datos que se presentan en los temas que le siguen, temas donde se asumirá que el lector no tiene problemas con la memoria dinámica en C++.

1.2 Vectores en memoria dinámica

En C++ podemos pedir memoria dinámica en tiempo de ejecución con los operadores **new** y **delete**. Recuerde que existen dos parejas de operadores:

- Pedir/liberar un objeto con los operadores **new** y **delete**; devuelve un puntero al objeto:

```
Tipo* puntero;
puntero = new Tipo;
//...  uso de *puntero
delete puntero;
```

  Esta reserva la podemos considerar como de un objeto individual —no un vector— sin olvidar que realmente el objeto *puntero* se puede manejar como un vector. Por ejemplo, puede pasar como puntero a un vector de tamaño 1 o podemos usar *puntero*[0].

- Pedir/liberar objetos con los operadores **new**[] y **delete**[]; devuelve un puntero al primer elemento de un vector de objetos:

```
int n= 10;
Tipo* puntero;
puntero = new Tipo[n];
//...  uso de puntero[0] a puntero[9]
delete[] puntero;
```

  Esta reserva la podemos considerar como de un vector de objetos. Este énfasis se justifica al imponer la necesidad de liberar con el operador **delete**[]; es un error liberar el bloque de memoria con el operador sin corchetes. Incluso si la reserva es de un objeto —n vale 1— la operación es distinta.

Son dos reservas diferenciadas; no se pueden mezclar. A pesar de ello, el puntero que obtiene la dirección del bloque de memoria reservado tiene exactamente las mismas características y se puede usar de la misma forma... excepto cuando lo liberemos.

1.2.1 Vectores de tamaño cero y punteros nulos

Un aspecto importante que aparece en el estándar y que puede simplificar el código se refiere a qué ocurre cuando se reserva un vector de tamaño cero o se libera un puntero nulo. Son dos operaciones que, en principio no tienen sentido, pero que se admiten en el lenguaje.

Vectores de tamaño cero

Es posible la reserva de un vector de tamaño cero. La reserva se realiza, aunque no se puede usar ningún espacio reservado. Por ejemplo, en el siguiente código:

```
int n=0;
int* puntero
puntero = new int[n];
//...  No se puede usar *puntero o puntero[0]
delete[] puntero;
```

se realiza una reserva, pero al tener cero elementos no es posible usar ni la primera posición del vector. Para ello, deberíamos haber reservado al menos un elemento.

Si no se puede usar el espacio, ¿para qué reservar? Efectivamente, no nos sirve el espacio reservado, pero nos permite crear código que sea válido incluso para tamaño cero. Por ejemplo, si crea un programa que reserva un vector de n elementos, no tendrá que realizar algo como:

```
unsigned int n;

std::cout << "Introduzca tamaño: ";
std::cin >> n;

unsigned int* puntero;
if (n==0)
   puntero = 0; // Si C++11: mejor nullptr
else puntero = new int[n];

for (unsigned int i=0; i<n; ++i)
   puntero[i]= 0;
```

puesto que también es válida la reserva de `0` elementos. Note que, en caso de reservar cero elementos, el bucle no llega a usar ninguna zona reservada.

Punteros nulos

Es posible llamar a liberar el espacio apuntado con el puntero nulo (`0`, *NULL* o **nullptr**). Realmente no se libera nada. No hay memoria apuntada, no hay que hacer nada. Podemos decir que lo primero que hace **delete**/**delete**[] es comprobar si es el puntero nulo, en cuyo caso, ignora la operación.

El resultado es que podemos simplificar código. Si nuestro programa usa un puntero que puede o no apuntar a una zona reservada, no es necesario tener cuidado con el puntero nulo. Por ejemplo, el código siguiente:

```
if (puntero != nullptr)
  delete[] puntero;
```

no aparecerá en nuestros programas, pues el puntero nulo no es un caso particular. En ese código, estamos realizando dos veces esa comprobación.

No sólo nuestros códigos se simplifican, sino que es más difícil equivocarse. Si no pudiéramos liberar el puntero nulo, olvidarnos de comprobar esa condición daría lugar a un error de ejecución.

Note que podemos usar el valor `0`, *NULL* o **nullptr**. Este último es válido sólo a partir de C++11. Es posible que se sienta más cómodo con los primeros si ha desarrollado código para C++98. En las siguientes secciones asumiremos que trabaja con C++11 o superior para que empiece a usar el valor recomendado **nullptr**.

1.2.2 Ejemplos de reservar/liberar estructuras de vectores

Para ejercitar la reserva y liberación de vectores vamos a trabajar con un ejemplo sencillo centrado en la reserva y liberación de vectores en memoria dinámica. Note que evitamos la definición de clases, centrándonos en el uso de estructuras simples que recuerden cómo funciona el código a este nivel.

Es importante recordar que el comportamiento de una clase es idéntico al de una estructura. La única diferencia es el acceso por defecto: en el caso de las clases es privado y en la estructuras público.

Ejemplo 1.2.1 Escriba un programa simple que lea un valor n y represente en memoria dinámica todos los valores que aparecen en el *triángulo de Pascal* de n filas.

En la figura 1.1 se puede ver cómo construir el triángulo de tamaño 6. Observe que las flechas indican los dos sumandos que dan lugar al siguiente valor.

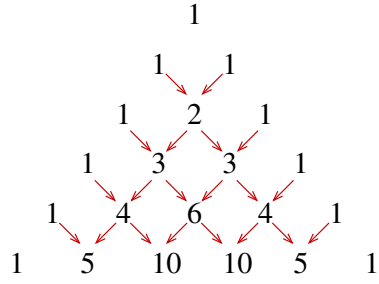


Figura 1.1
Triángulo de Pascal de tamaño 6.

Definición 1.1 — Triángulo de Pascal. Un triángulo de Pascal de tamaño n es una estructura bidimensional con n filas de forma que la fila i contiene i columnas. El valor de la fila f columna c es el número combinatorio $\binom{f}{c}$, siendo f y c valores que comienzan en cero.

En nuestro ejemplo, vamos a crear/destruir una estructura en memoria dinámica que contenga los valores de n filas. Para ello, tendremos en cuenta la forma de creación que indica la figura 1.1, es decir, la fila f contiene:

- Si estamos en el primer o último lugar, un valor 1. Es decir, en las columnas 0 y f.
- En los valores intermedios, la posición j se calcula como la suma de las columnas $j-1$ y j de la fila anterior.

que se almacenarán en una estructura con el siguiente diseño:

```
struct TrianguloPascal {
  // ... representación en memoria dinámica
  int n;        // número de filas del triángulo
};
```

En las siguientes secciones desarrollamos distintas alternativas para la representación del triángulo en memoria dinámica.

Reserva de un único vector

La estructura más simple es un único bloque, un vector que contiene todos los elementos del triángulo almacenados por filas una a continuación de la otra. El código de la estructura `Triangulo` para esta representación es el siguiente:

```
struct TrianguloPascal {
  int* datos;
  int n;        // número de filas del triángulo
};
```

Para crearla, tenemos que calcular el número total de posiciones necesarias: si tenemos un triángulo de f filas, tendrá $1+2+3+\cdots+f$ elementos.

Es tentador crear un bucle que acumule el tamaño de las filas, pero recordemos que esta suma corresponde a una *progresión aritmética*[1] de sumandos s_i que se puede resolver en una simple expresión. En concreto, la suma vale:

$$\sum_{i=1}^{n} s_i = n \cdot \frac{s_1 + s_n}{2} = f \cdot \frac{1+f}{2}$$

En la figura 1.2 se muestra gráficamente cómo quedarían almacenados los elementos en el vector. Observe que aparecen muchos valores 1, pues están situados en los extremos de las filas.

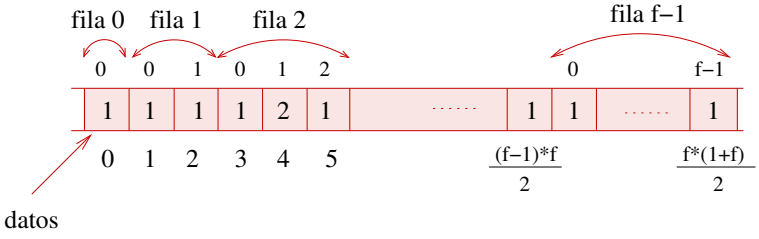


Figura 1.2
Representación con un único vector.

En nuestro programa, podemos calcular el número de elementos de un triángulo con la siguiente función:

```
int NumeroDatos (int filas) {
   return filas<=0 ? 0 : (filas * (filas+1)/2);
}
```

[1] La diferencia entre un sumando y el siguiente es una constante.

Ejercicio 1.1 En la función anterior se usa la expresión de una progresión aritmética. Suponga que cambiamos el orden de los operandos como sigue:

```
int NumeroDatos (int filas) {
   return filas<=0 ? 0 : ((filas+1)/2 * filas);
}
```

¿Qué ocurre si usamos esta nueva expresión?

Las funciones para crear es la siguientes:

```
void Crear (TrianguloPascal& t, int filas) {
   if (filas <= 0) {
      t.datos= nullptr;
      t.n= 0;
   }
   else {
      t.n= filas;
      t.datos= new int[NumeroDatos(filas)];
   }
}
```

Note que hemos impuesto que el puntero `t.datos` vale cero en caso de que el triángulo no tenga filas. Sin embargo, la siguiente solución también es válida:

```
void Crear (TrianguloPascal& t, int filas) {
   t.n= filas;
   t.datos= new int[NumeroDatos(filas)];
}
```

puesto que la función `NumeroDatos` devuelve un cero y la reserva de cero elementos también es válida.

Por otro lado, la liberación del espacio se puede realizar como sigue:

```
void Destruir (TrianguloPascal& t) {
   delete[] t.datos;
   t.n= 0;
   t.datos= nullptr;
}
```

donde liberamos el puntero `t.datos` sin comprobar si vale puntero nulo. Es correcta en ambos casos, si vale puntero nulo o si ha reservado cero elementos. Si hacemos:

```
void Destruir (TrianguloPascal& t) {
   if (t.n != 0) {
      delete[] t.datos;
      t.n= 0;
      t.datos= nullptr;
   }
}
```

el resultado no sería el mismo. Si `t.datos` vale cero, es correcto. Si hemos reservado un vector de tamaño cero, no lo es. Aunque se reserve un tamaño cero, no evitamos la necesidad de liberar el espacio con una operación **`delete[]`**.

Reserva de un vector de vectores

La facilidad con la que podemos acceder al elemento de la fila f, columna c del triángulo sugiere el diseño que proponemos en esta sección. La idea es crear un vector de punteros para apuntar a cada fila. Se muestra gráficamente en la figura 1.3.

La implementación implica la siguiente definición:

```
struct TrianguloPascal {
   int* * datos;
   int n;
};
```

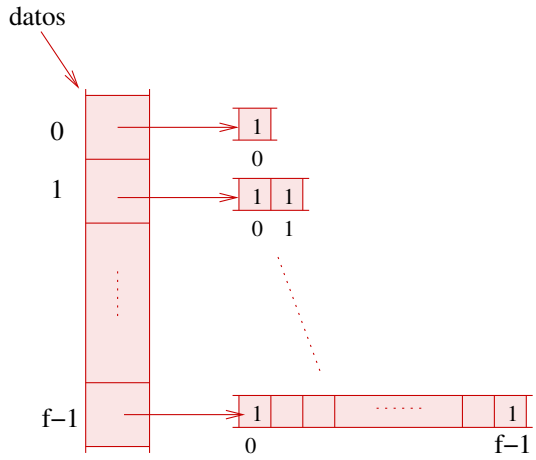


Figura 1.3
Representación con punteros a vectores.

donde podemos ver que `datos` es un puntero que apunta a objetos de tipo **`int`**`*`, es decir, a punteros que a su vez apuntarán a cada vector fila. Con este código, podemos acceder al elemento en `f,c` con una expresión tan simple como `t.datos[f][c]`.

La función para reservar el espacio necesario puede ser la siguiente:

```
void Crear(TrianguloPascal& t, int filas) {
   if (filas <= 0) {
      t.datos= nullptr;
      t.n= 0;
   }
   else {
      t.n= filas;
      t.datos= new int*[filas];
      for (int i=0; i<filas; ++i)
         t.datos[i]= new int[i+1];
   }
}
```

Note que hemos diferenciado el caso de triángulo vacío asignando un puntero nulo a `datos`. Observe el número de operaciones **`new`**`[]` que son necesarias: la del bloque de punteros más una para cada fila.

La correspondiente función para liberar deberá aplicar una operación **`delete`**`[]` a cada bloque reservado. Un posible código es:

```
void Destruir(TrianguloPascal& t) {
   for (int i=0; i<t.n; ++i)
      delete[] t.datos[i];
   delete[] t.datos;
   t.n= 0;
   t.datos= nullptr;
}
```

Ejercicio 1.2 Implemente una función, por ejemplo `Rellenar`, que recibe una estructura de tipo `TrianguloPascal` como la creada con la función `Crear` anterior y rellena los valores correspondientes al *triángulo de Pascal*.

Reserva de un vector de punteros a un vector

En esta sección proponemos una solución que mezcla las dos anteriores. Tiene la ventaja de poder usar el doble corchete para acceder a una posición y simplifica la reserva de memoria, pues sólo tenemos que reservar dos bloques. Se muestra gráficamente en la figura 1.4.

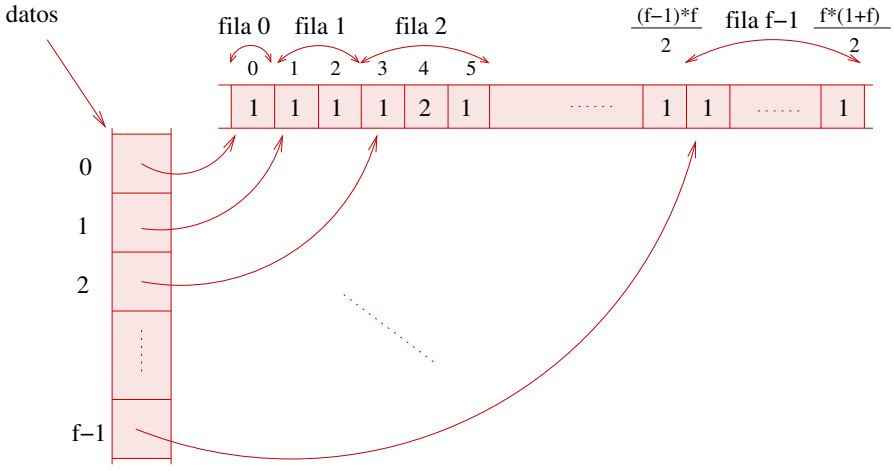


Figura 1.4
Representación con punteros a un vector.

La implementación implica la siguiente definición:

```
struct TrianguloPascal {
   int* * datos;
   int n;
};
```

En este caso sólo tenemos que realizar dos operaciones **new**[], una para el bloque de punteros y otra para el bloque con todos los elementos. El código es:

```
void Crear(TrianguloPascal& t, int filas) {
   if (filas <= 0) {
      t.datos= nullptr;
      t.n= 0;
   }
   else {
      t.n= filas;
      t.datos= new int*[filas];
      t.datos[0]= new int[NumeroDatos(filas)];
      for (int i=1; i<t.n; ++i)
         t.datos[i]= t.datos[i-1]+i;
   }
}
```

donde hemos aprovechado, de nuevo, la función `NumeroDatos` que calcula el número de datos de un triángulo.

La función para liberar es la siguiente:

```
void Destruir(TrianguloPascal& t) {
   if (t.datos) // Recuerde, equivale a t.datos!=0
      delete[] t.datos[0];
   delete[] t.datos;
   t.n= 0;
   t.datos= nullptr;
}
```

Note que tenemos que comprobar si el puntero `datos` vale cero para no hacer `t.datos[0]`.

Ejercicio 1.3 Implemente una función *Imprimir* que muestre el triángulo en la salida estándar, una fila en cada línea. Una vez finalizada, compruebe que también es válida para la representación anterior.

Fin ejemplo 1.2.1 ■

Vector vs vector de vectores

¿Qué representación sería más conveniente? En muchos textos encontrará una justificación a favor del vector de punteros por la conveniencia en la sintaxis de doble corchete. En C++ es muy simple resolver este problema, pues podríamos crear una función *Elemento*[2] que nos simplifica el acceso:

```
int& Elemento(TrianguloPascal& t, int f, int c) {
   return (t.datos + NumeroDatos(f)) [c];
}
const int& Elemento(const TrianguloPascal& t, int f, int c) {
   return (t.datos + NumeroDatos(f)) [c];
}
```

aunque implica que tenemos que hacer un cálculo para localizar la posición de la fila. A pesar de ello, no tiene por qué ser más lento, pues el cálculo es bastante simple y se realiza con una instrucción simple del procesador, mientras que el acceso mediante puntero a puntero implica una indirección a una zona de memoria para saltar a otra.

Ejercicio 1.4 Implemente una función *Rellenar* que recibe una estructura como la siguiente:

```
struct TrianguloPascal {
  int* datos;
  int n;
};
```

ya creada con la función *Crear* y rellena los valores correspondientes al *triángulo de Pascal*.

Por otro lado, las operaciones **new**/**delete** son también costosas. En este sentido, es más conveniente un simple vector. Además de que es una única reserva, los elementos están en posiciones adyacentes, lo que facilita el tiempo de ejecución cuando recorremos el triángulo[3].

Desde este punto de vista, la versión de múltiples vectores por fila resulta poco recomendable. En la práctica, podría ser una buena opción en el caso en que tuviéramos que intercambiar o realizar operaciones por filas, que no es el caso.

Por valor o por referencia: interface vs representación

Cuando creamos una función que recibe un objeto de tipo *TrianguloPascal* debemos decidir cómo se pasa: ¿por valor o por referencia? En principio, la respuesta parece muy simple, pues sólo tenemos que decidir si cambia o no.

Sin embargo, en los ejemplos anteriores podemos plantear una duda: el paso por valor implica la copia de la estructura, es decir, del puntero y el entero. Por ejemplo, si implementamos la función *Rellenar* del ejercicio 1.4 anterior, podemos proponer dos cabeceras:

```
void Rellenar(TrianguloPascal t);   // Paso por valor
void Rellenar(TrianguloPascal& t);  // Paso por referencia
```

[2] En la práctica, si se crea una clase, sobrecargaremos el operador paréntesis para simplificar la sintaxis.

[3] Recuerde que la arquitectura de un ordenador está normalmente optimizada para mejorar el tiempo de acceso a memoria en posiciones cercanas.

En este caso, nos preguntamos: ¿el triángulo `t` cambia? En las implementaciones anteriores, no, puesto que la reserva ya está hecha. Por tanto, si escogemos cualquiera de las representaciones anteriores, las dos alternativas funcionarán.

A pesar de ello, debemos recordar que las cabeceras de las funciones establecen la interfaz, no deben definirse desde el punto de vista de la representación, al menos, no solamente. La idea es que la cabecera de la función `Rellenar` es parte de la interfaz; en el primer caso expresa una copia del objeto mientras que en el segundo expresa cambios en el objeto. Por ejemplo, imagine que hace una función `Calcular` que se supone que no cambia el triángulo y que puede tener esta forma:

```
void Calcular (const TrianguloPascal& t) {
   // ...
   Rellenar(t);
   // ...
}
```

Si usamos la versión paso por valor, el compilador no generará ningún error. Todo es correcto, aunque puede tener efectos inesperados, pues esa función realmente modifica el triángulo. En cambio, si usamos la versión por referencia, el compilador dirá que esta función `Calcular` no es correcta.

Otro caso más claro sobre la conveniencia de la segunda versión es que la representación se haga sin memoria dinámica. Por ejemplo, imagine que sabemos que nuestro triángulo tendrá como mucho 6 filas. Podemos proponer:

```
struct TrianguloPascal {
  int datos[21];
  int n;
};
```

Ahora no hay duda, es necesaria la versión paso por referencia. En realidad, la versión paso por valor era funcional pero incorrecta, pues arrastraba una dependencia sobre la representación: necesitaba que los datos se almacenaran "fuera de la estructura".

Este ejemplo nos da la oportunidad de recordar que las decisiones sobre los pasos/devoluciones por valor, por referencia o por referencia constante son una forma de establecer una interfaz. El compilador comprobará cómo se realizan y nos ayudará a detectar inconsistencias.

Cuando se implemente un nuevo tipo de dato se conocen los detalles internos; no se obsesione con el detalle sobre si funciona o no, también está creando una interfaz que deberá ser válida cuando cambie la representación.

Por otro lado, también se puede plantear qué ocurre cuando se devuelve una estructura por valor, es decir, cuando usamos **`return`** de una estructura. En los ejemplos anteriores, con punteros, tendremos una copia muy simple, de un puntero y un entero. Sin embargo, con la última representación de tamaño fijo, no funciona igual.

Si a todo esto añadimos la asignación, la posibilidad de que un puntero sobre-escriba a otro —que podría apuntar a zona reservada— todo se complica. Si queremos trabajar con este nuevo tipo de dato y lo resuelve con una estructura al estilo C, nuestros programas serán más dependientes de la representación. Afortunadamente, vamos a trabajar con clases en C++, donde el lenguaje ya ofrece soluciones para resolver todas estas situaciones.

Ejercicio 1.5 Implemente la pareja de funciones `Elemento` válida para las representaciones de vector de punteros a las filas. Úselas para escribir una función `Imprimir` —válida para las tres representaciones— que muestra el triángulo formateado en la salida estándar.

1.2.3 Almacenamiento automático vs dinámico

Podemos distinguir el almacenamiento *estático* y *dinámico*. El primero corresponde a memoria para variables globales que no cambia durante la ejecución, mientras que el segundo cambia. En

este sentido, tanto la pila como el montón corresponden a zona dinámica. Sin embargo, en la práctica hablaremos de memoria automática para la pila y dinámica para referirnos a la zona del montón.

El almacenamiento de objetos en la pila —*stack*— o en el montón —*heap*— da lugar a soluciones muy distintas. Algunos detalles a tener en cuenta son:

- El almacenamiento en la pila es muy rápido. El programa no tiene más que preparar la zona de memoria que hay en el tope, básicamente moviendo la posición de éste tanto espacio como requiera el nuevo objeto.
- El almacenamiento en el montón requiere de operaciones **new**/**delete** que pueden ser muy costosas. Recuerde que la zona de memoria dinámica se debe gestionar para controlar el espacio libre y ocupado, con reserva y liberación en cualquier orden, lo que da lugar a un mapa de memoria particionado con múltiples huecos libres y ocupados.
- La zona de memoria automática en la pila puede estar limitada. En general, la pila puede incluso ser una zona de memoria relativamente pequeña, puesto que no está diseñada para almacenar grandes cantidades de datos, sino para gestionar la reserva y liberación automática que resulta de la ejecución de nuestro programa.

La mayoría de nuestras soluciones se basarán en la gestión de memoria dinámica. La creación de estructuras de datos complejas dará lugar a objetos de un tamaño que crece y decrece hasta tamaños relativamente grandes o limitados por la memoria disponible. Para optimizar el uso de la memoria, la única opción será la reserva en memoria dinámica.

No por ello debemos olvidar la posibilidad de soluciones en memoria estática/automática, pues resultan especialmente útiles cuando el espacio requerido es pequeño y las restricciones de tiempo de ejecución son importantes. Por ejemplo, si tiene que crear un programa de respuesta rápida en tiempo real, no debería dejar el resultado en manos del sistema de memoria dinámica.

Un ejemplo de uso de la memoria automática con un tamaño adaptado a la ejecución son los *arrays de longitud variable —VLA, variable length array*, en inglés— que permiten reservar memoria en la pila para un tamaño indeterminado en compilación. Por ejemplo:

```
#include <iostream>
int main() {
   int n;
   std::cin >> n;

   double datos[n]; // Error, ¡no es C++ estándar!
   // ...
```

Este tipo de reserva es válida en C, y por ello algunos compiladores pueden aceptarla como una extensión en C++. No está aceptado en el estándar, por tanto, no lo admitiremos como una solución. Aunque esas extensiones puedan sugerir que se incorporará, los inconvenientes para añadirlo son cada vez mayores, no sólo porque es peligroso reservar mucho espacio en la pila —puede hacer que el programa falle— sino porque no es fácil integrarlo en el lenguaje, especialmente desde los últimos estándares donde se trabaja intensivamente en aumentar la capacidad del compilador para procesar código en tiempo de compilación.

Vector y biblioteca estándar

Si nuestra solución es la memoria dinámica, tendremos que usar **new**/**delete**. Sí y no. Si tiene que crear un programa con memoria dinámica, piense en primer lugar en la biblioteca estándar. Por ejemplo, si queremos crear el tipo `TrianguloPascal`, una buena solución sería:

```
struct TrianguloPascal {
  std::vector<int> datos;
};
```

Nada más. Sí, es memoria dinámica porque el tipo **std::vector** reserva en memoria dinámica, pero nosotros podemos trabajar con este vector que resuelve la gestión del recurso de

manera automática. No hay fallos de reserva, no hay fallos en la liberación, no hay detalles de copias de punteros y no de datos, no hay fallos de asignación. Todo es correcto, porque la estructura aprovecha las operaciones bien definidas del tipo **std::vector**.

Además, si su programa funcionaba en C++98 y ahora lo recompila en C++11, obtendrá una versión del tipo **vector** más optimizada, sin hacer ningún cambio.

En la práctica, usar *punteros "desnudos"* será algo poco habitual en un programa de alto nivel. Es posible que tenga que usarlos si está creando una biblioteca a bajo nivel y tiene que implementar los detalles de una representación optimizada, o si tiene que llamar a rutinas de más bajo nivel —de C, por ejemplo— pero la mayoría de los problemas no los necesitan. Incluso si quiere manejar un objeto en memoria dinámica para el que no necesita un contenedor, usará un puntero *"inteligente"* —del inglés *smart pointer*— que también ofrece la biblioteca estándar.

Una vez entendida la conveniencia de la biblioteca estándar para la mayoría de sus futuros programas, recordemos que estamos estudiando los detalles de la creación de representaciones de estructuras de datos que crecen/decrecen dinámicamente. Es parte del objetivo de este libro que resolvamos el problema a bajo nivel, lo que implica necesariamente soluciones para practicar con gestión dinámica de memoria.

Cadenas de la biblioteca estándar

El tipo **std::string** es parte de la biblioteca estándar y nos permite manejar cadenas de caracteres con gran comodidad. Sus similitudes con el tipo **std::vector** y la discusión sobre el almacenamiento automático/dinámico nos invitan a reflexionar sobre la representación de este tipo de dato.

El tipo **std::string** contiene una secuencia de objetos **char**. Las instancias de este tipo darán lugar a muchos casos en los que el tamaño tan pequeño sugiere que una representación en memoria automática sería mucho más eficiente. Sin embargo, sabemos que un objeto puede crecer hasta tamaños muy grandes. Por tanto, es inevitable optar por una representación en memoria dinámica.

En la práctica, se puede proponer una solución intermedia que puede encontrar en la bibliografía como *SSO*, del inglés *small/short string optimization*. La representación es, básicamente, unir en la misma zona de memoria las dos opciones:

- Si quiere una representación automática declararía un vector de caracteres de objetos **char** con cierto tamaño. Suponga que usamos esta opción si la cadena tiene hasta 16 caracteres.
- Si quiere una representación en memoria dinámica necesita, al menos, un puntero que apunte a la zona reservada.

Esto lo podemos hacer con una unión como sigue:

```
class Cadena {
   union {
      char buffer[16];
      char* puntero;
   };
   int longitud; // size_type en std::string
   // otros componentes comunes
public:
   // ...
```

donde hemos definido una **union** anónima, es decir, los nombres `buffer` y `puntero` están disponibles directamente en el ámbito de `Cadena`. Recuerde que como **union**, implica que los dos están situados en la misma zona de memoria, es decir, se usará uno u otro.

Dependiendo de la longitud, usaremos la versión con `buffer` o la reserva en memoria dinámica controlada con `puntero`. Esto permite trabajar con cadenas cortas sin que haya necesidad de ninguna operación **new**/**delete**.