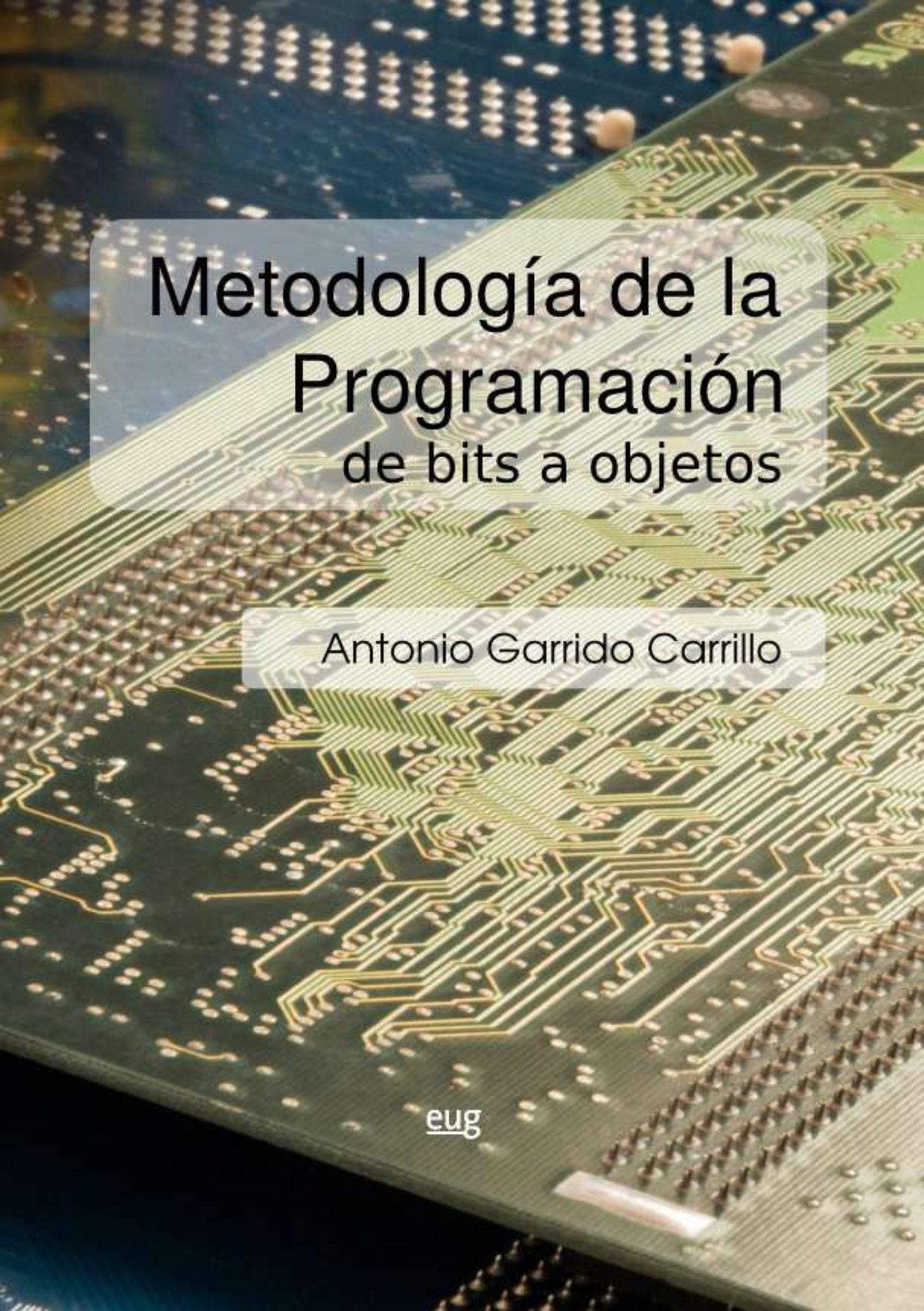

Metodología de la Programación
de bits a objetos
Antonio Garrido Carrillo
eug

METODOLOGÍA DE LA PROGRAMACIÓN

de bits a objetos

ANTONIO GARRIDO CARRILLO

METODOLOGÍA DE LA PROGRAMACIÓN

de bits a objetos

GRANADA
2016

© ANTONIO GARRIDO CARRILLO
© UNIVERSIDAD DE GRANADA
METODOLOGÍA DE LA PROGRAMACIÓN: de bits a objetos
ISBN: 978-84-338-5870-2
Edita: Editorial Universidad de Granada.
Campus Universitario de Cartuja. Granada.

Printed in Spain *Impreso en España*.

Cualquier forma de reproducción, distribución, comunicación pública o transformación de esta obra sólo puede ser realizada con la autorización de sus titulares, salvo excepción prevista por la ley.

Índice general



1 Representación en binario 1

1.1 Introducción 1

1.2 Bases binaria, octal y hexadecimal 1

1.2.1 De base 10 a base b 2

1.2.2 Cambio agrupando dígitos 4

1.2.3 Números con parte fraccionaria 5

1.3 Enteros y reales en el ordenador 6

1.3.1 Representación de enteros 6

1.3.2 Representación de reales 8

1.4 Arquitectura del ordenador 10

1.4.1 Datos en memoria 11

1.5 Caracteres 12

1.5.1 Código ASCII: Extensiones 13

1.5.2 Unicode 14

1.6 Problemas 16

2 Tipos básicos: Bits 17

2.1 Introducción 17

2.1.1 Tipos y lenguaje de alto nivel 17

2.2 Tipos aritméticos 18

2.2.1 Tipo booleano 18

2.2.2 Tipos carácter 19

2.2.3 Tipos enteros . . . 19
2.2.4 Tipos en coma flotante . . . 20
2.2.5 Sinónimos con typedef . . . 21
2.3 Características de los tipos 22
2.3.1 Tamaños. El operador *sizeof* . . . 23
2.3.2 Límites numéricos . . . 23
2.4 Tipo enumerado 24
2.4.1 Representación del tipo enumerado . . . 25
2.5 Conversiones 26
2.5.1 Conversiones implícitas . . . 27
2.5.2 Conversiones explícitas . . . 29
2.6 Tipos de datos y bits 31
2.6.1 Operadores a nivel de bit . . . 32
2.6.2 Campos de bits . . . 35
2.6.3 Uniones . . . 36
2.7 Problemas 39

3 Vectores y matrices . . . 41
3.1 Introducción 41
3.2 Vectores 41
3.2.1 Declaración . . . 42
3.2.2 Vector de constantes . . . 44
3.2.3 Límites de un vector . . . 45
3.3 Paso de vectores a funciones 46
3.3.1 Vectores de constantes . . . 48
3.3.2 Tamaño delimitado por *centinela* . . . 52
3.4 Ejemplos: Búsqueda y ordenación 52
3.4.1 Algoritmos de búsqueda . . . 52
3.4.2 Algoritmos de ordenación . . . 54
3.5 Cadenas de caracteres 56
3.5.1 Paso de cadenas a funciones . . . 57
3.5.2 E/S de cadenas . . . 58
3.5.3 Cadenas de caracteres y tipo *string* . . . 60
3.6 Matrices 61
3.6.1 Declaración . . . 62
3.6.2 Almacenamiento y límites de matrices . . . 62
3.6.3 Paso de matrices a funciones . . . 63
3.7 Arrays multidimensionales 70
3.7.1 Paso de arrays a funciones . . . 71
3.8 Problemas 72

4 Estructuras y vectores-C 75

4.1 Introducción 75

4.2 Definición de estructuras 76

4.2.1 Declaración de variables 76

4.2.2 Definición en C y typedef 77

4.2.3 El operador punto 78

4.3 Mezclando estructuras y vectores-C 78

4.3.1 Asignación y copia de estructuras con vectores 81

4.3.2 Estructuras como miembro 84

4.4 Problemas 88

5 Punteros 89

5.1 Introducción 89

5.2 El tipo de dato *Puntero a T* 89

5.2.1 Operaciones básicas 91

5.2.2 Puntero a *void* 93

5.3 Estructuras 94

5.3.1 Puntero a un campo de la estructura 95

5.3.2 Punteros miembro a la propia estructura 96

5.3.3 Declaración adelantada 97

5.4 Funciones 98

5.4.1 Devolución de punteros a objetos locales 99

5.5 Vectores 100

5.5.1 Aritmética de punteros 101

5.6 Punteros y const 103

5.6.1 Punteros, funciones y const 105

5.6.2 Punteros, vectores y const 105

5.6.3 typedef y const 106

5.7 Cadenas de caracteres 107

5.8 Problemas 108

6 Memoria dinámica 113

6.1 Introducción 113

6.2 Estructura de la memoria 113

6.3 Reservar y liberar memoria dinámica 115

6.3.1 Reserva y liberación de espacio para un objeto 115

6.3.2 Reserva y liberación de vectores 119

6.3.3 Gestión automática de memoria 122

6.4 Ejemplos de estructuras de datos simples 122
6.4.1 Celdas enlazadas . . . 122
6.4.2 Vectores y Matrices . . . 126
6.4.3 Cursores . . . 129
6.5 Problemas 131

7 Funciones . . . 135
7.1 Introducción 135
7.2 La función *main* 135
7.2.1 Otras formas de finalización de *main* . . . 137
7.2.2 Parámetros de main y tipo *string* . . . 138
7.3 La Pila 139
7.3.1 Estructura de la memoria . . . 139
7.3.2 Variables locales a un bloque . . . 141
7.3.3 Resolviendo llamada y retorno . . . 142
7.4 Paso parámetros/devolución de resultados 143
7.4.1 Paso/devolución en C . . . 144
7.4.2 Paso/devolución en C++. Tipo referencia . . . 147
7.4.3 Parámetros y *const* . . . 153
7.4.4 Ejemplo de paso de parámetros . . . 155
7.5 Nuevas características 156
7.5.1 Funciones *inline* . . . 156
7.5.2 Sobrecarga de funciones . . . 157
7.5.3 Parámetros con valor por defecto . . . 159
7.5.4 Variables locales *static* . . . 159
7.6 Punteros a función 160
7.6.1 Nombrando el tipo puntero a función . . . 164
7.7 Problemas 165

8 Errores . . . 169
8.1 Introducción 169
8.1.1 Errores de compilación y ejecución . . . 170
8.1.2 Errores de origen endógeno y exógeno . . . 171
8.2 Localizar y depurar errores 174
8.2.1 Compilación vs ejecución . . . 174
8.2.2 Errores lógicos . . . 176
8.2.3 Invariantes . . . 177
8.2.4 Localización de errores . . . 180
8.3 Mecanismos de gestión de errores 184
8.3.1 ¿Qué errores gestionar? . . . 185
8.3.2 Sistemas clásicos de gestión . . . 188
8.3.3 Excepciones . . . 192

8.3.4 Excepciones estándar . . . 205
8.4 Problemas 208

9 Clases: conceptos básicos . . . 209
9.1 Introducción 209
9.1.1 Abstracción . . . 209
9.1.2 Usando abstracciones . . . 210
9.2 Diseño basado en la abstracción 212
9.2.1 El problema: sumar matrices . . . 212
9.2.2 Una solución directa sin módulos . . . 213
9.2.3 Abstracción funcional . . . 214
9.2.4 Abstracción de datos . . . 217
9.3 Diseño e implementación de clases simples 221
9.3.1 Ocultamiento de información . . . 221
9.3.2 Definición de una clase . . . 224
9.3.3 Separación en archivos . . . 235
9.4 Encapsulamiento: la clase *Matriz* 238
9.4.1 Representación en base a un vector de tamaño fijo . . . 238
9.4.2 Implementación con la *STL* . . . 246
9.5 Problemas 247

10 Clases: diseño . . . 249
10.1 Introducción 249
10.2 Diseñar la parte interna 249
10.2.1 Función de abstracción . . . 250
10.2.2 Invariante de la representación . . . 250
10.2.3 Métodos privados . . . 251
10.3 Diseñar la interfaz 252
10.3.1 Selección de las operaciones . . . 252
10.3.2 Niveles de encapsulación . . . 253
10.3.3 La interfaz implementa la abstracción . . . 257
10.3.4 Atributos públicos . . . 258
10.3.5 Orden de las secciones de una clase . . . 259
10.4 Atributos y métodos de clase 259
10.4.1 Miembros static constantes . . . 262
10.4.2 Atributos/métodos de clase vs variables/funciones globales . . . 263
10.5 Clases como atributos 265
10.5.1 Clases y métodos amigos . . . 266
10.5.2 Construir los atributos . . . 266
10.6 Diseño de nuevos tipos 269
10.6.1 Ejemplos de clases . . . 269
10.6.2 Estructuras y clases . . . 273

10.7 Problemas **275**

11 Clases: avanzado 277

11.1 Introducción **277**
11.1.1 Comportamiento como tipos predefinidos 277

11.2 Funciones básicas de una clase **279**
11.2.1 Construcción y destrucción 281
11.2.2 Copia y asignación 284
11.2.3 Funciones miembro generadas por defecto 291

11.3 Construcción, destrucción y asignación en detalle **294**
11.3.1 Llamadas a constructores y destructores 294
11.3.2 Aprovechando constructor y destructor en la asignación 302
11.3.3 Ejemplos de llamadas generadas 304
11.3.4 Eficiencia de las operaciones 309

11.4 Definiciones dentro de la clase **313**
11.4.1 Clases dentro de clases 314

11.5 Excepciones **316**
11.5.1 Clases y rebobinado de la pila 316
11.5.2 Excepciones vs devolución de errores 317
11.5.3 Código seguro ante excepciones 318

11.6 Problemas **326**

12 Sobrecarga de operadores 329

12.1 Introducción **329**
12.1.1 Sobrecarga de funciones/operadores 330

12.2 Mecanismo de sobrecarga de operadores **330**
12.2.1 Sobrecarga como función externa 330
12.2.2 Sobrecarga como función miembro 331
12.2.3 Operadores como funciones miembro o externas 332

12.3 Sobrecargando operadores **333**
12.3.1 Operador de indexación 334
12.3.2 Operadores de asignación 337
12.3.3 Operadores relacionales 339
12.3.4 Operadores de incremento y decremento 340
12.3.5 Operadores de E/S 342
12.3.6 Operador de *llamada a función* 345
12.3.7 Ejemplo: la clase *Complejo* 349

12.4 Problemas **354**

13 E/S. Acceso aleatorio 355

13.1 Introducción **355**

13.2 Flujos asociados a ficheros 356
13.2.1 Flujos y buffers . . . 357
13.2.2 Ficheros de lectura y escritura . . . 358
13.2.3 Apertura y cierre de archivos . . . 358
13.3 Acceso secuencial y aleatorio 361
13.3.1 Secuencial vs aleatorio . . . 362
13.3.2 E/S de bloques de caracteres . . . 363
13.3.3 Operaciones de posicionamiento . . . 365
13.4 Ficheros binarios y de texto 367
13.4.1 Conversiones de caracteres especiales . . . 368
13.4.2 Ficheros binarios que contienen texto . . . 371
13.4.3 Ficheros binarios, E/S por bloques y acceso aleatorio . . . 372
13.5 Archivos, registros y campos 378
13.5.1 Registros de longitud fija o variable . . . 378
13.5.2 Operaciones con registros . . . 379
13.5.3 E/S de objetos de una clase . . . 383
13.6 Flujos asociados a *string* 386
13.7 Problemas 389

14 C++11/14 . . . 391
14.1 Introducción 391
14.2 Tipos fundamentales y enumerados 392
14.2.1 Enumerados con ámbito . . . 393
14.2.2 Nuevos literales numéricos en *C++14* . . . 393
14.3 Nuevos tipos carácter, cadenas y *string* 394
14.3.1 Literales de tipo carácter . . . 394
14.3.2 Literales cadena de caracteres . . . 396
14.3.3 Tipos *string* . . . 398
14.4 Semántica de movimiento 401
14.4.1 lvalues y rvalues en C++98 . . . 402
14.4.2 Lvalues y rvalues en C++11/14 . . . 406
14.5 Clases en C++11/14 413
14.5.1 Funciones miembro generadas por defecto . . . 413
14.5.2 Constructores . . . 416
14.6 Algunos aspectos adicionales 419
14.6.1 Especificación de excepciones . . . 419
14.6.2 Expresiones constantes en compilación . . . 420
14.6.3 Literales de usuario . . . 421

A Solución a los ejercicios . . . 423
A.1 Representación binaria 423
A.2 Tipos básicos: Bits 425

A.3 Vectores y matrices 431
A.4 Estructuras y vectores-C 441
A.5 Punteros 444
A.6 Memoria dinámica 452
A.7 Funciones 455
A.8 Errores 459
A.9 Clases: conceptos básicos 463
A.10 Clases: diseño 468
A.11 Clases: avanzado 473
A.12 Sobrecarga de operadores 482
A.13 E/S. Acceso aleatorio 492
A.14 C++11/14 499

B Tablas 503
B.1 Tabla ASCII 503
B.2 Operadores C++ 504
B.3 Palabras reservadas de C89, C99, C11, C++ y C++11 506

Bibliografía 509

Índice alfabético 513

Prólogo



El objetivo de este libro es presentar conceptos de *Metodología de la programación* como un curso básico de programación. No debería considerarse un curso sobre un lenguaje concreto —en nuestro caso C++— sino una exposición principalmente motivada por conceptos que un estudiante de primer curso debería conocer.

Un futuro profesional debería conocer la forma en que se puede crear un sistema que resuelve un problema partiendo de las operaciones más básicas que nos ofrece el hardware: cómo podemos ascender de las operaciones más simples que implementa un ordenador hasta un diseño complejo que es capaz de resolver, de adaptarse y de mejorar de una forma simple y eficaz a pesar de su complejidad.

Este libro ofrece un curso que permite a un estudiante de primeros cursos empezar a comprender esta metodología, preparándolo para abordar cursos más avanzados y especializados que permitan completar una formación integral para enfrentarse a problemas de desarrollo software de mayor tamaño.

Aprender a programar: Contexto del curso

Aprender a programar no es una tarea simple. Aunque muchos cursos o tutoriales se oferten con mensajes del tipo *"aprenda a programar en X días"*, la formación de un programador debe pasar por distintos cursos que le ocuparán mucho más tiempo, desde cursos básicos hasta cursos más avanzados y especializados.

Este libro corresponde a un segundo curso de programación. Se supone que el estudiante ya ha asimilado unos contenidos básicos de programación donde se incluyan los conceptos fundamentales sobre programación estructurada, tipos de datos simples y compuestos, incluso una introducción a la compilación separada (véase la referencia Garrido[14]). Con esos conocimientos, el estudiante ya es capaz de resolver problemas simples y de entender cómo funciona un programa, cuál es el flujo de control de un programa y cómo se pueden abstraer algunos algoritmos en módulos.

Este segundo curso es de profundización. Ahora se estudiarán en profundidad los detalles más básicos de los tipos, es decir, mostraremos detalles más cercanos a la máquina, pero se alcanzarán

niveles de abstracción mucho más altos. El estudiante incluirá temas más complejos, como la reserva de recursos en memoria dinámica, pero se enfrentará a ellos proponiendo soluciones que permitan disponer de tipos y módulos que abstraen los detalles importantes para formular las soluciones fácilmente a pesar de la complejidad de los programas.

Como resultado final de este curso, el alumno debería entender mejor cómo la metodología de la programación nos permite enfrentarnos a problemas de mayor tamaño creando soluciones eficaces, simples de desarrollar y de modificar. Lógicamente, serán necesarios otros cursos para completar la formación como programador. Sin embargo, las bases obtenidas después de este curso lo sitúan en un nivel de conocimiento ideal para empezar a abordar contenidos más avanzados y especializados. Algunos de los aspectos que podemos destacar son:

- El estudiante ha trabajado con tipos de datos básicos y ha profundizado en el problema de la codificación, la representación de información en memoria o la memoria dinámica. Estos aspectos facilitan estudiar temas relacionados con programación a bajo nivel, más cercana al sistema operativo o incluso que interactúe con sistemas hardware para los que la API se formule en términos muy cercanos al hardware.
- El estudiante ha trabajado en profundidad el problema del encapsulamiento. Es uno de los pilares de la programación dirigida a objetos. Es el momento ideal para ampliar la capacidad de diseño incluyendo temas de herencia o polimorfismo. Además, se han incluido discusiones sobre la forma en que se usa la jerarquía de clases que ofrece el lenguaje para flujos, por lo que el estudiante debería haber comenzado a asimilar la facilidad o conveniencia de este tipo de programación.
- Se han propuesto distintas soluciones para los mismos problemas, incluyendo comentarios que se refieren a la eficiencia en memoria y en tiempo. Por ejemplo, se comentan distintos algoritmos de búsqueda y ordenación. Es un buen momento para entrar en detalle en el problema de diseñar algoritmos eficientes. Ya sabe lo importante que puede ser cambiar un algoritmo, ahora debería aprender a diseñar mejores algoritmos.
- La abstracción de datos muestra claramente cómo podemos ocultar la representación de un tipo de dato mediante una interfaz. Una de las decisiones que hay que tomar es la elección de la representación del tipo. Al ser un curso básico, se ha trabajado sólo con representaciones sencillas basadas fundamentalmente en vectores. Es el momento ideal para proponer nuevas formas de resolver los problemas, como son los árboles u otras estructuras no lineales.
- Se ha enfatizado continuamente la idea de la abstracción como una herramienta fundamental. Se habla de abstracción funcional, se enfatiza la abstracción de la representación incluso en los tipos simples, se crean clases que ofertan una interfaz para abstraer los aspectos relevantes del nuevo tipo. Es muy sencillo e incluso natural abordar nuevas formas de abstracción, como la programación genérica.

Podríamos incluir otros comentarios, pues es fácil enlazar las implicaciones de un curso básico de programación con cursos más avanzados. Sin embargo, basta con este conjunto de aspectos para poner de relieve los conceptos básicos que se presentan en este libro, así como la conveniencia como base ideal para garantizar los mejores resultados en los cursos más avanzados.

Con estas consideraciones, este libro podría situarse como una segunda parte de un curso básico de fundamentos de programación, o un curso intermedio que pone los cimientos para poder abordar con garantías temas más avanzados y especializados de programación.

El lenguaje de programación

Los contenidos del libro se desarrollarán sobre el lenguaje de programación C++. No se debe interpretar que está diseñado para enseñar este lenguaje, aunque la amplia variedad de situaciones y discusiones nos permite interpretar el libro como una exposición razonada de las herramientas que

nos ofrece este lenguaje. No se mostrarán todos los detalles, aunque sí las partes más importantes y habituales que le permitirán empezar a usarlo para resolver sus propios problemas.

Lógicamente, el lenguaje C++ será un componente fundamental del libro. Inevitablemente, gran parte del contenido se dedicará a entender los detalles sintácticos y las particularidades del lenguaje, pero no deja de ser una forma de asimilar y afianzar los conocimientos que se pretenden adquirir durante el curso. Al finalizar, el estudiante debería estar preparado tanto para desarrollar soluciones en este lenguaje como para adaptarse a uno nuevo.

La elección de este lenguaje como vehículo para alcanzar los objetivos de este libro no es casual. La capacidad que C++ tiene para expresar soluciones a muy bajo nivel hasta diseños a un nivel de abstracción muy alto —incluyendo diseños basados en la programación dirigida a objetos, programación genérica o metaprogramación— lo convierten en una herramienta ideal para poder describir cómo el desarrollo del software nos permite avanzar desde las soluciones más simples hasta complejos sistemas. El estudiante puede estudiar cómo resolver problemas que van desde los bits y la gestión de recursos muy cercana al sistema hasta discusiones a nivel de jerarquías de clases sin necesidad de cambiar de lenguaje.

Esta amplia gama de discusiones no se hace a un bajo costo. Efectivamente, el lenguaje puede parecer complejo, e incluso poco recomendable para un primer curso. Sin embargo, los objetivos del curso no son simples de conseguir y, aunque le parezcan ambiciosos, resultan ideales como una base donde apoyarse para cursos más avanzados de programación. Para compensar esta dificultad, este libro no hace una exposición guiada solamente por el lenguaje, sino siguiendo un esquema más pedagógico asumiendo que el estudiante es relativamente novato. Aprovechamos el lenguaje para situar una más amplia base de formación para el estudiante, aunque el curso resulte algo más complejo. Note que otras referencias bibliográficas hacen una exposición mucho más exhaustiva —véase Josuttis[17] o Stroustrup[37]— recorriendo todos los aspectos del lenguaje y que son ideales para programadores formados interesados en aprender un nuevo lenguaje en su totalidad.

Los estudios universitarios de Ingeniería Informática

La razón principal que ha motivado la creación de este libro ha sido la necesidad de ofrecer un documento práctico que facilite tanto la docencia del profesor como el aprendizaje del alumno en los cursos de programación de los estudios de *Ingeniería Informática*. La raíz de esta necesidad se sitúa en la dificultad de encontrar referencias bibliográficas que se adapten a los planes de estudios.

La evolución de los planes de estudios en los últimos 30 años ha desembocado en un modelo de enseñanza en el que el estudiante tiene una gran responsabilidad: por un lado, se han ampliado los contenidos de la carrera, mientras que por otro se han disminuido el tiempo de enseñanza en el aula. Una parte del esfuerzo debería realizarse de forma autodidacta. Los estudiantes deben asimilar una gran cantidad de contenidos en poco tiempo, por lo que resulta muy difícil disponer de un buen material de estudio si se limita únicamente a los apuntes de clase. Por ello, es necesario disponer de algún libro de apoyo.

Trabajar los contenidos de la asignatura a partir de varios libros implica un sobreesfuerzo que no es simple para un alumno de primer curso. Un libro que se adapte a los contenidos de la asignatura es ideal para obtener el mejor resultado, dejando las consultas bibliográficas en libros especializados para temas más puntuales, así como el modelo de trabajo autónomo para cursos más avanzados.

Como resultado, el libro constituye una buena referencia no sólo para el estudiante de estos estudios, sino para el lector que quiera introducirse en el mundo de la programación de forma autodidacta.

Organización del libro

Los capítulos se han organizado para que el lector vaya asimilando desde conceptos de más bajo nivel hasta los más abstractos. Los primeros temas le permitirán crear soluciones sencillas con los tipos más básicos del lenguaje, pero reconociendo los detalles de bajo nivel más cercanos a la máquina. En los últimos no sólo estarán ocultos los detalles más básicos, sino que desaparecerán los detalles de las estructuras de datos que el alumno diseñe.

Cada tema introduce algunos aspectos adicionales para que el lector empiece a utilizarlos, pero pueden ser completados y asimilados completamente a lo largo de ese y otros capítulos posteriores. Por tanto, cuando se aborda un tema concreto del lenguaje no tiene por qué estudiarse en su totalidad, sino que puede retrasar algunos aspectos más avanzados para temas posteriores. Recuerde: no recorremos el lenguaje, sino que vamos avanzando en los conceptos de programación que requieren poco a poco de más herramientas del lenguaje.

En el **capítulo 1** se introduce al lector en el mundo de la representación y la codificación. La mayoría de estos detalles no se usan explícitamente en los programas que normalmente se realizan, pero son fundamentales para entender el porqué del comportamiento de algunos programas. En cursos posteriores, descubrirá que muchos de estos contenidos se dan por entendidos porque se consideran básicos para cualquier programador.

En el **capítulo 2** se establece la conexión entre los tipos básicos del lenguaje y la representación expuesta en el capítulo anterior. Aunque haya creado múltiples soluciones con los tipos más básicos, es importante conocer la relación que hay entre los datos que usamos y la forma en que se representan en la máquina. Después de este tema debería tener muy clara la potencia que ofrecen los tipos más básicos, pero también las limitaciones, así como la simplicidad con la que una simple representación mediante bits permite poder representar una amplia gama de tipos de datos.

En el **capítulo 3** se presentan los tipos de datos compuestos, concretamente los vectores y matrices. En los temas anteriores el lector debe haber asimilado en detalle en qué consisten los tipos más simples. En este tema, se avanza en la complejidad de los datos introduciendo nuevos tipos y relacionándolos con los conocimientos adquiridos. Estas herramientas permiten presentar ejemplos basados en algoritmos clásicos de búsqueda y ordenación. Es importante que note que los tipos expuestos corresponden a los que ya ofrece el lenguaje C. Realmente, el capítulo se dedica al estudio de los vectores a más bajo nivel. Esto nos permite estudiarlos de una forma muy cercana a la máquina, como una secuencia de tipos que se almacenan en posiciones consecutivas de memoria. De nuevo, a pesar de ser una estructura de datos más compleja, el lector puede entender cómo hemos pasado de una representación más básica a una más potente. A pesar de su bajo nivel, ya podemos hablar de matrices multidimensionales o cadenas de caracteres.

En el **capítulo 4** se presentan las estructuras como un tipo de dato compuesto heterogéneo. Esta forma de crear nuevos tipos de datos compuestos completa las herramientas básicas para los tipos compuestos: vectores y estructuras. Motivado primero por la simplicidad de las estructuras como composición de tipos más simples y segundo por establecer la relación con el capítulo anterior, este tema se completa con contenido que presenta la relación que existe entre estructuras y *vectores-C*, pues lo habitual será proponer soluciones que mezclen ambas formas de composición de tipos.

En el **capítulo 5** se presenta el tipo de dato puntero. Aunque es muy simple, la cantidad tan amplia de posibilidades y relaciones con los temas anteriores lo hacen bastante complejo. Es necesario dedicarle bastante tiempo para que el lector asimile correctamente todas las posibilidades, por lo que se ha dedicado este tema completo. Los conocimientos adquiridos en los temas previos, especialmente relativos a vectores, son fundamentales para manejar correctamente el tipo puntero.

En el **capítulo 6** se estudia el sistema de gestión dinámica de memoria. En los temas anteriores se han presentado prácticamente todas las herramientas necesarias, a falta de los conceptos directamente relacionados con memoria dinámica. Por ello, este tema resulta más sencillo y puede

completarse con una breve introducción a algunas estructuras de datos básicas que justifican la importancia de la memoria dinámica e introduce al lector en el estudio de las estructuras de datos.

En el **capítulo 7** se vuelven a visitar las funciones. El este libro se asume que el estudiante ya tiene conocimientos básicos sobre funciones. No sólo eso, sino que en los temas anteriores ya se han estado usando. Sin embargo, el correcto manejo de las funciones constituye un pilar fundamental en el desarrollo de programas en C++, no sólo para este libro, sino para temas más avanzados. En este sentido, este tema vuelve a insistir sobre estos conocimientos, ampliando con algunos conceptos que no habían aparecido —sobrecarga, valores por defecto, etc.— y estudiando en mayor detalle lo que el lector ya ha aprendido.

Tenga en cuenta que cuando se comienza a trabajar con funciones, el estudiante no necesita más que una idea relativamente intuitiva de abstracción funcional, sin necesidad de entender por qué funcionan y qué detalles se tienen que resolver a bajo nivel. Después de este tema, estos conocimientos deberían quedar claros. Además, se incluyen los tipos puntero a función que permiten que el estudiante vea las funciones de una forma más real y lo preparen para cursos más avanzados.

En el **capítulo 8** se estudia el problema de la gestión de errores. A estas alturas, el estudiante ya ha trabajado muchos problemas e incluso formulado programas que disponen de distintos módulos. Probablemente ya haya abordado el problema de gestión de errores simples y conozca —al menos intuitivamente— la necesidad de establecer una forma eficaz de gestionarlos. Con los conocimientos que ha adquirido hasta el tema anterior, ya es capaz de entender distintas alternativas para resolverlo, e incluso comprender en qué consiste la gestión basada en excepciones, un mecanismo avanzado pero que se puede entender a la perfección si conoce los detalles de cómo funcionan los programas.

En el **capítulo 9** se da un paso importante en la formación: el estudio de clases. Hasta ahora, los problemas se resolvían centrando la modularización en la abstracción funcional, en la solución de algoritmos y la creación de módulos capaces de trabajar con distintos datos. En este capítulo se presenta el problema de la abstracción en la creación de módulos para gestionar nuevos tipos que permiten ocultar detalles de representación ofreciendo una interfaz más simple. Es un concepto complicado para alguien sin experiencia, por lo que el tema se centra en exponer los conceptos básicos. Lo importante, en este punto, es que el lector conozca la idea y la conveniencia de expresar las soluciones incorporando la abstracción y el encapsulamiento.

En el **capítulo 10** se estudia más ampliamente el problema del diseño basado en clases. Esta forma de crear soluciones suele ser muy complicada para programadores sin experiencia, ya que el diseño es uno de los aspectos más delicados y para los que se requiere más experiencia. En el tema se incluye un estudio más detallado de las posiblidades que ofrecen las clases, incluyendo la posibilidad de crear soluciones con múltiples clases relacionadas.

En el **capítulo 11** se completa el bloque de temas dedicado al estudio de clases. En este caso, se estudian en profundidad. Se abordan todos los aspectos de su funcionamiento, incluyendo representaciones complejas que incluyen gestión de recursos a bajo nivel, en nuestro caso, memoria dinámica. Realmente después de este tema es cuando el estudiante es capaz de crear cualquier clase en C++, sabiendo en detalle cómo funciona y cómo resolver los problemas que puede presentar si no funciona. Además, se incluyen algunas discusiones más avanzadas sobre diseño de clases y cómo hacer que la gestión de errores sea más simple, es decir, obtener soluciones más seguras y fáciles de mantener.

En el **capítulo 12** se presenta el problema de la sobrecarga de operadores. Es un tema con mucho contenido sintáctico y detalles del lenguaje. Sin embargo, es un tema de especial interés, primero porque profundiza en la abstracción y segundo porque obliga al estudiante a entender con mucho más detalle aspectos concretos del lenguaje. Por ejemplo, sirve para recordar cómo funciona el paso y devolución por referencia o incluso cómo funcionan los operadores sobre los tipos básicos definidos en el lenguaje. Si bien no es un tema fundamental para un curso de metodología de la

programación, sí se justifica como un tema importante para poder estudiar temas más avanzados del lenguaje y reforzar algunos conceptos que se han presentado en temas anteriores.

En el **capítulo 13**, se completa el curso sobre C++98 con contenidos relacionados con la E/S. El estudiante ya es capaz de entender muchos aspectos del sistema de E/S de C++, pero no lo conoce por completo. Una vez que ha recorrido los temas anteriores, está preparado para entender cómo funcionan los tipos que ofrece el lenguaje.

Se estudia cómo podemos trasladar la información que codificamos en memoria a disco, así como las ideas más básicas sobre el acceso secuencial y aleatorio que nos ofrecen los dispositivos externos. Todos estos conceptos constituyen un primer paso para introducirse en los problemas de almacenamiento en *bases de datos*.

Además, se presentan los tipos de flujos asociados a **`string`**, de manera que las jerarquías que se presentan sirven como una primera aproximación y motivación para el estudio de temas relacionados con la programación dirigida a objetos.

Finalmente, en el **capítulo 14** se completa el curso con contenidos referidos a C++11/14. No podemos olvidar que los futuros programadores de C++ trabajarán en el estándar actual. Además, si empieza a revisar código de terceros o quiere participar en el desarrollo de algún proyecto en C++, conviene actualizar el contenido a este nuevo estándar.

Además, se incluyen varios apéndices que completan los temas anteriores:

- En el **apéndice A** aparecen las soluciones a los ejercicios propuestos a lo largo del libro. Para que el lector asimile más fácilmente los contenidos, se proponen ejercicios a la largo de los temas, de forma que el estudio de éstos se realice de una forma más interactiva. Cuando se propone un ejercicio, el lector debería intentar resolverlo proponiendo su solución, tras lo que debería consultar la solución propuesta en este apéndice.
- En el **apéndice B** se presentan algunas tablas de interés para un programador de C++.

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.

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 continuamente cambia e incorpora nuevos conceptos. 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, buffer, precondición, endianness, puntero, log, desreferenciar, argumento, miembro, serialización,* etc.

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* o *array* en lugar de *arreglo*.

Además, es probable que haya alguna otra palabra. Por ejemplo, la palabra *sumatoria* que encuentro en la *RAE* como *sumatorio*. ¿Por qué la hacen masculina si *suma* es femenina? Que yo recuerde, todos mis profesores y compañeros la han llamado y la llaman *sumatoria*.

Sin duda, mis cambios son 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 certificado mi respeto a la lengua y a la RAE como institución, 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.

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

Finalmente, quiero agradecer el apoyo de los compañeros que han usado este material como guía en sus propias clases, ya que con ello me han animado a completarlo y a terminar este libro.

A. Garrido.
Febrero de 2016.

1 Representación en binario

Introducción 1
Bases binaria, octal y hexadecimal 1
Enteros y reales en el ordenador 6
Arquitectura del ordenador 10
Caracteres 12
Problemas 16

1.1 Introducción

La unidad de información más básica y a partir de la que vamos a diseñar nuestras aplicaciones es el *bit*, que representa la información necesaria para distinguir entre dos estados: el *0* y el *1*. Así, la representación interna de los datos se realiza en base a ceros y unos. Esta representación binaria para cada tipo de dato puede variar de un sistema a otro. En general, no será necesario conocer los detalles de un sistema para poder programar sobre él. Sin embargo, es importante conocer en qué consiste para entender mejor el comportamiento de nuestros programas.

Los tipos fundamentales que puede usar en sus programas y que es interesante estudiar en detalle son los siguientes:

- Tipos integrales. Incluyen el tipo booleano, los caracteres y los números enteros.
- Tipos en coma flotante. Permiten representar un número real.

Antes de estudiar cómo podemos representar estos tipos de datos, es fundamental entender la notación numérica en distintas bases, especialmente en base binaria y decimal.

1.2 Bases binaria, octal y hexadecimal

La base decimal es la que estamos acostumbrados a usar cada día. Para expresar un número en base 10 necesitamos 10 dígitos: *01234546789*. Recordemos que una forma sencilla de entender cómo funciona es expresar los números enteros que se van generando cuando contamos de uno en uno: cuando sumamos uno a las unidades, cambiamos al dígito siguiente; si el dígito es *9* —el último— volvemos a cero y añadimos uno a las decenas. Esta idea generalizada a un número cualquiera de dígitos nos permite generar cualquier valor entero.

Una descripción más formal incluyendo dígitos decimales se puede realizar considerando que el valor de un número expresado en base decimal, por ejemplo *2915.407*, se puede expresar como una suma de términos basados en potencias de 10. En nuestro ejemplo:

$$\mathbf{2}\cdot 10^3+\mathbf{9}\cdot 10^2+\mathbf{1}\cdot 10^1+\mathbf{5}\cdot 10^0+\mathbf{4}\cdot 10^{-1}+\mathbf{0}\cdot 10^{-2}+\mathbf{7}\cdot 10^{-3}$$

La misma mecánica se puede aplicar a cualquier número n en base b distinta a la decimal (que denotaremos como n_b). Por ejemplo, si queremos trabajar con base octal (b=*8*) sólo tenemos que definir los 8 dígitos que usaremos y plantear la misma solución. Los dígitos pueden ser *01234567*. Contar de uno en uno es ir cambiando de dígito en las unidades y cuando llegamos al 7 —el último— volvemos al cero y añadimos uno al dígito de la izquierda —las *"octenas"*—. Así, el número $10_{b=8}$ corresponde a ocho unidades.

Realmente, en la práctica no hay diferencias entre distintas bases y se pueden realizar exactamente las mismas operaciones. Por ejemplo, podemos realizar una descripción más general para expresar un número con parte fraccionaria en base octal. El número 5543.3203_{b8} se puede expresar como:

$$\mathbf{5}\cdot 8^3+\mathbf{5}\cdot 8^2+\mathbf{4}\cdot 8^1+\mathbf{3}\cdot 8^0+\mathbf{3}\cdot 8^{-1}+\mathbf{2}\cdot 8^{-2}+\mathbf{0}\cdot 8^{-3}+\mathbf{3}\cdot 8^{-4}$$

Esta descripción ya sugiere el método que usaremos para transformar cualquier número en una base b a base 10: calcularemos el valor de la sumatoria usando las potencias de b. Observe que este cálculo puede resultar especialmente fácil de realizar para pasar de base 2 a base 10. En este caso los dígitos son el cero y el uno, por lo que la sumatoria se transforma en la suma de las potencias que corresponden al valor 1. En la figura 1.1 se muestra un ejemplo.

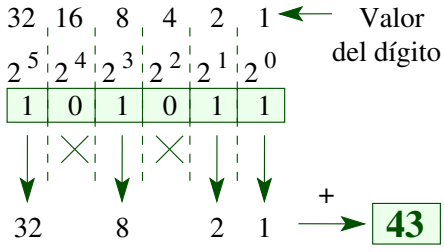


Figura 1.1
Paso de base 2 a base 10.

Observe que para realizar la sumatoria, sólo tenemos en cuenta los unos ya que los ceros corresponden a productos que se anulan. El resultado es que sólo es necesario sumar los valores $32+8+2+1$ obteniendo el número 43_{b10}.

Ejercicio 1.1 Transforme a base 10 el número binario 1111_{b2}.

1.2.1 De base 10 a base b

La división entera —sin decimales— de un número entero entre 10 nos permite obtener un resto que corresponde a las unidades (el dígito más a la derecha) y un cociente que corresponde al resto del número (todos los dígitos eliminando el de las unidades). Este algoritmo es una forma simple de extraer cada uno de los dígitos que componen un número en base 10. Si queremos conocer los dígitos en cualquier otra base podemos aplicar el mismo procedimiento, aunque dividiendo por la base correspondiente.



Ejemplo 1.2.1 Transformar el número 86139_{b10} a base 8.

La solución consiste en ir dividiendo sucesivamente por 8. En la primera división (86139/8) obtenemos un cociente de 10767 y un resto de 3. Por tanto, el dígito menos significativo de la solución será 3. Recuerde que esto es exactamente lo mismo que hacemos en base 10: cuando calculamos el resto entre 10, obtenemos el valor de las unidades.

Dividiendo entre 8 sucesivamente hasta que obtenemos un cociente de cero obtenemos una secuencia de restos que corresponden a cada uno de los dígitos. En la figura 1.2 se muestra gráficamente.

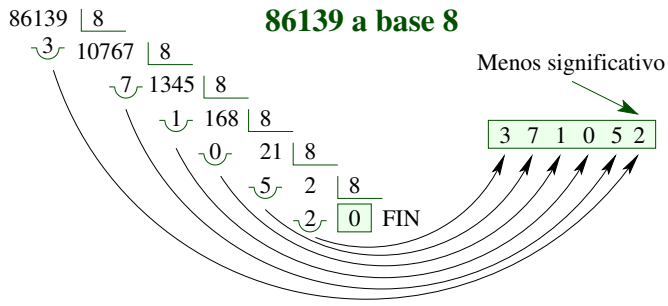


Figura 1.2
Paso de base 10 a base 8.

El fin del procedimiento se alcanza al obtener un cociente de cero. Observe que la disposición de los dígitos en el resultado es contraria al orden de cálculo de éstos: el primero obtenido —las unidades— es el último en la solución.

Por tanto, la solución al problema es:

$$86139_{b10} = 371052_{b8}$$

Fin ejemplo 1.2.1 ■

Ejemplo 1.2.2 Transformar el número 10443325_{b10} a base 16.

La novedad en este caso es que un número escrito en base 16 no puede ser expresado sólo con los 10 dígitos *0123456789*. Tenemos 16 dígitos, y por tanto necesitamos poder expresar los 6 valores restantes con un sólo dígito. Para resolver el problema, seleccionamos una forma de escritura para todos estos dígitos. Lo más simple: usar las primeras letras del alfabeto. Por tanto, los dígitos serán: *0123456789ABCDEF*

El algoritmo para pasar un número es el mismo: realizar divisiones sucesivas —ahora por 16— hasta llegar al cero. La sucesión de restos serán valores del 0 al 15. Los restos del 10 al 15 se representan con los nuevos dígitos numéricos. En la figura 1.3 podemos ver la transformación.

Observe que los restos *{10, 11, 12, 13, 14, 15}* se expresan, respectivamente, con las letras *ABCDEF*. En nuestro ejemplo: el resto 15 se expresa con la letra *F*, el resto 10 con *A* y el resto *13* con *D*.

Por tanto, la solución al problema es:

$$10443325_{b10} = 9F5A3D_{b16}$$

Fin ejemplo 1.2.2 ■

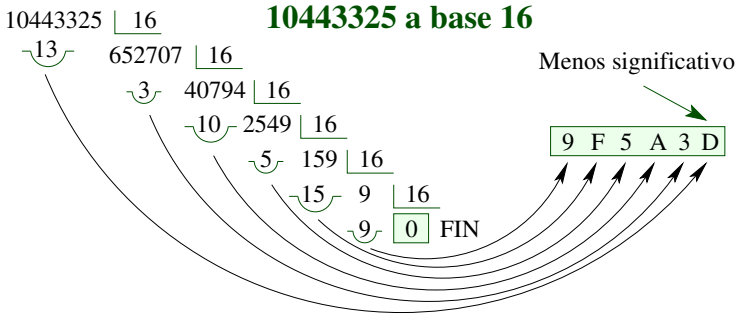


Figura 1.3
Paso de base 10 a base 16.

Ejercicio 1.2 Transformar el número 43_{b10} a base 2.

1.2.2 Cambio agrupando dígitos

En programación es importante conocer cómo funciona la representación binaria, ya que es la base sobre la que se desarrolla la tecnología de nuestros ordenadores. ¿Por qué hablar de base 8 o 16? El motivo es que son bases muy relacionadas con la primera, ya que ambas son potencias de 2: $8 = 2^3$ y $16 = 2^4$.

Para entenderlo mejor, imagine que queremos pasar un número en base 10 a base 100. Para realizarlo, como hemos visto en la sección anterior, deberíamos ir dividiendo entre 100 para extraer la secuencia de restos que componen el número. Esta división es muy simple, ya que la división entera entre 100 da como resultado un resto que corresponde a los dos últimos dígitos y un cociente que corresponde al resto de dígitos. Lógicamente, las sucesivas divisiones van extrayendo el resto de dígitos de dos en dos. Esta facilidad es consecuencia de que 100 es potencia de 10 ($100 = 10^2$). Note que si usáramos una base mayor, por ejemplo 10^3, las divisiones obtendrían los dígitos en grupos de 3.

Esta misma facilidad se puede usar para los números en base 2. Un número en base 2 puede llegar a ser muy largo. Por ejemplo, números del orden de pocos miles se tienen que expresar con secuencias de 10 o más dígitos binarios. Además, sólo aparecen los dígitos 0 y 1, por lo que es más propenso a errores. Por tanto, trabajar manualmente con números binarios es costoso y peligroso.

Para resolver el problema se suele trabajar con una base potencia de 2: en base 8 y 16. La transformación de un número de base 2 a cualquiera de estas bases es muy sencilla. Lo único que tenemos que hacer es agrupar los números en grupos de 3 o 4, dependiendo de la base.

En la tabla 1.1 se muestran algunos valores y sus correspondientes números en binario, octal y hexadecimal. Para las representaciones octal y hexadecimal se han añadido entre paréntesis las agrupaciones correspondientes.

Los ejemplos son números que necesitan como mucho de 8 bits: un byte. Observe la comodidad que nos facilita esta representación:

- Notación *octal*. En este caso resulta muy fácil establecer una correspondencia entre un dígito octal (0-7) y su representación binaria. Es muy fácil recordar la tripleta correspondiente a cada dígito para pasar de una base a otra.
- Notación *hexadecimal*. Este caso resulta algo más complicado pues no estamos acostumbrados a usar letras en los dígitos, y es más difícil recordar las 16 cuádruplas. Sin embargo, resulta muy conveniente al agrupar un byte en dos dígitos hexadecimales. Esta notación es incluso más habitual que la octal, ya que los números binarios que encontramos sue-



Tabla 1.1
Bases decimal/binaria/octal/hexadecimal

Decimal	binario	Octal	hexadecimal
0	00000000	000 (00-000-000)	00 (0000-0000)
1	00000001	001 (00-000-001)	01 (0000-0001)
2	00000010	002 (00-000-010)	02 (0000-0010)
...	...	...	...
8	00001000	010 (00-001-000)	08 (0000-1000)
...	...	...	...
14	00001110	016 (00-001-110)	0E (0000-1110)
15	00001111	017 (00-001-111)	0F (0000-1111)
16	00010000	020 (00-010-000)	10 (0001-0000)
...	...	...	...
32	00100000	040 (00-100-000)	20 (0010-0000)
33	00100001	041 (00-100-001)	21 (0010-0001)
...	...	...	...
254	11111110	376 (11-111-110)	FE (1111-1110)
255	11111111	377 (11-111-111)	FF (1111-1111)

len tener una tamaño múltiplo de 8, y pueden hacerse corresponder con parejas de dígitos hexadecimales.

1.2.3 Números con parte fraccionaria

Recordemos que el cambio de base de un número en base b a base 10 es muy simple, ya que se puede hacer expresando el número como la sumatoria correspondiente en base a las potencias. Por ejemplo, el número 207.46_{b8} en base 8 se puede transformar a base 10 como el resultado de la siguiente expresión:

$$\mathbf{2}\cdot 8^2+\mathbf{0}\cdot 8^1+\mathbf{7}\cdot 8^0+\mathbf{4}\cdot 8^{-1}+\mathbf{6}\cdot 8^{-2}=135.59375_{b10}$$

Por otro lado, la transformación de base 10 a otra base de números con parte fraccionaria se puede realizar en dos fases: (a) resolviendo la parte entera como se ha visto en las secciones anteriores y (b) resolviendo la parte fraccionaria.

Para cambiar de base un número que está en el rango $(0,1)$, es decir que tiene sólo parte decimal, multiplicamos sucesivamente por la base. Por ejemplo, si tenemos un número en base 10 y queremos extraer el primer dígito, podemos multiplicar por 10 de forma que el dígito de las décimas pasa a ser las unidades. Lógicamente, todos los dígitos decimales se mueven una posición a la izquierda.

Ejemplo 1.2.3 Transforme el número 135.59_{b10} a base 8.

En primer lugar transformamos 135 como se ha visto anteriormente, obteniendo 207_{b8}. Para transformar 0.59 será necesario realizar multiplicaciones sucesivas por 8. Cada vez que multiplicamos por 8 obtenemos:

- En la parte entera un nuevo dígito de la parte decimal en base 8.
- En la parte fraccionaria el resto del número que corresponde a los dígitos que quedan por transformar.

En la figura 1.4 puede ver gráficamente cómo hemos ido obteniendo diferentes dígitos de la parte decimal del número 0.59_{b10}. Observe que hemos obtenido un número con 5 dígitos pero el proceso aún puede continuar. En base 10 teníamos solamente 2 dígitos, en otra base incluso podríamos tener un número de dígitos infinito.

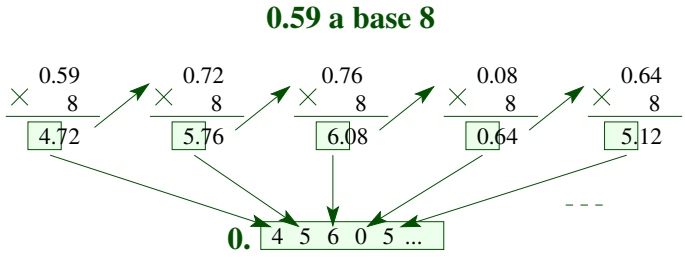


Figura 1.4
Parte fraccionaria de base 10 a base 8.

En nuestro ejemplo podemos decir que:

$$135.59_{b10} \approx 207.45605_{b8}$$

Fin ejemplo 1.2.3 ■

Ejercicio 1.3 Pase el número 10.6_{b10} de base 10 a base 2.

1.3 Enteros y reales en el ordenador

En teoría un entero o un real puede llegar a tener un número infinito de dígitos. En el ordenador, donde el espacio es limitado y las operaciones se deben realizar de forma eficiente se establece un tamaño fijo de dígitos binarios para representar un número. Por ejemplo, podemos hacer que un número se represente con 32 bits. Establecer un tamaño fijo de dígitos binarios para representar un número tiene como consecuencias:

1. *Podemos desperdiciar memoria*. Por ejemplo, para representar el cero sólo necesitamos un dígito. Si usamos 32 bits estamos desperdiciando la mayoría de ellos.
2. *No se pueden representar todos los números*. Si usamos 32 bits tenemos solamente 2^{32} posibles números distintos. Tenemos un número finito de posibilidades para representar un número infinito de números.

Para resolver el problema tenemos que establecer una forma de codificación de números. En concreto, una para los enteros y otra para los reales.

1.3.1 Representación de enteros

El problema de representar un entero tiene una codificación natural al tener una correspondencia directa entre una secuencia de bits y su valor binario. Por ejemplo, si disponemos de 32 bits podemos representar los valores del rango 0 al 4294967295. Estos valores son las 2^{32} codificaciones posibles: desde el 0 al $2^{32} - 1$. Si tuviéramos un número de bits distinto tendremos un rango distinto, por ejemplo si sólo disponemos de 8 bits, tendremos $2^8 = 256$ posibles valores enteros: desde el 0 al 255.

El problema es que la representación anterior es válida para valores enteros no negativos. Si queremos disponer de datos positivos y negativos tendremos que codificarlo de otra forma. El método más simple que se nos puede ocurrir es dedicar un bit para codificar el signo. Por ejemplo,



si tenemos 8 bits, podemos dedicar el primero al signo —0 es positivo y 1 es negativo— y 7 bits para codificar el valor absoluto del entero —desde el 0 al 127—. Con esta codificación hemos perdido la mitad del rango de los enteros, aunque ahora podemos usar valores negativos. En la figura 1.5 se puede ver un ejemplo de codificación con 8 bits.

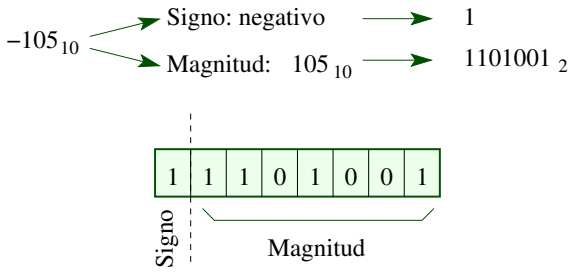


Figura 1.5
Codificar −105 como signo y magnitud.

Una codificación mucho más habitual de enteros con signo es la que se denomina *complemento a 2*. En esta codificación los valores positivos se codifican igual, es decir, el primer bit es cero y el resto corresponde al entero almacenado. Para los valores negativos se almacena el *complemento a 2*. El complemento a 2 de un número binario positivo consiste en:

1. Calcular el complemento a 1. Se cambian los unos por los ceros[1].
2. Calcular el complemento a 2. Se suma uno al resultado anterior.

En la figura 1.6 se puede ver un ejemplo de codificación del número −105 en complemento a 2. Observe que en este caso hemos codificado el número con 16 bits.

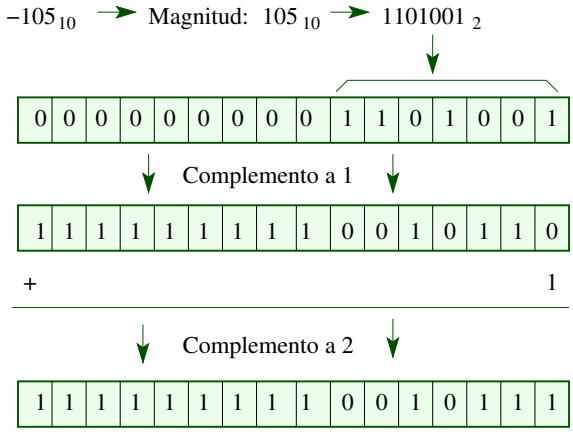


Figura 1.6
Codificar −105 en complemento a 2.

La ventaja del complemento a 2 es que facilita la implementación en hardware de las operaciones con números negativos. Además, nos permite representar un número más:

- La codificación con signo y magnitud produce dos representaciones para el cero: +0 y -0.
- La codificación en complemento a 2 nos permite representar un valor negativo adicional. En caso de 8 bits, el valor −128, o en el caso de 32 bits el valor −2147483648.

[1] Realmente es una operación de complemento a la base menos 1.

Ejercicio 1.4 Represente el valor -128 en complemento a 2 con 8 bits. Una vez obtenido, represente el valor 127 en binario y opere el resultado sumándole uno. ¿Qué valores obtiene?

Como resultado, un número entero con signo representado con 32 bits estará en el rango: $[-2147483648, 2147483647]$.

Es importante notar cómo la representación de números enteros nos permite implementar algoritmos que calculan los valores exactos siempre que no nos salgamos del intervalo de valores representados. Lógicamente, si realizamos una operación que obtiene un resultado fuera del rango de posibilidades de nuestros datos enteros, el resultado es indeterminado.

Finalmente, es interesante indicar que se han presentado estas dos representaciones, la primera por su simplicidad y la segunda por su amplio uso. Sin embargo, no debe entender que éstas son las dos posibilidades a las que se puede limitar el hardware. Cuando implementamos en un lenguaje de alto nivel no podemos garantizar nada sobre cuál es la representación de los datos.

1.3.2 Representación de reales

La representación de números reales es un problema más complejo. En este caso queremos representar muchos más valores, incluyendo los enteros. Además, sería conveniente poder representar también valores muy grandes o muy pequeños, positivos o negativos.

Un ejemplo de representación ampliamente usada y conocida es el **estándar del IEEE para aritmética en coma flotante *IEEE754***. En nuestro caso no nos interesan todos los detalles de cómo funciona este estándar, pero es interesante mostrar algunos aspectos que nos permitan entender el comportamiento de la aritmética con este tipo de datos.

En *IEEE754* se establecen varias longitudes con las que se podría representar un número en coma flotante[2], aunque nosotros nos centramos en la precisión simple —con 32 bits— para entender el formato.

El primer paso para codificar un número es transformarlo para expresarlo en el siguiente formato:

$$signo(1.b_1 b_2 \cdots b_n) \cdot 2^{exponente}$$

que corresponde a un número transformado a base 2 con parte fraccionaria, pero donde hemos movido la coma para situarla justo después del primer uno. Lógicamente, si movemos la coma para ponerla después de 1, tendremos que multiplicar por una potencia de 2[3].

Una vez tenemos el número normalizado con ese formato, ya podemos codificarlo. Para ello, se divide el espacio disponible para codificar —en nuestro caso 32 bits— en 3 partes:

1. *Signo*. En este bit se codifica si el número es negativo (0) o positivo (1).
2. *Exponente*. Reservamos 8 bits para el exponente. Como éste puede ser negativo, primero lo desplazamos sumando 127. Por ejemplo, si el exponente es -50, se almacena el número 77 (en base 2), o si el exponente es 20, se almacena el número 147.
3. *Mantisa*. Reservamos 23 bits para la mantisa. Corresponde a las 23 primeras posiciones del número normalizado, es decir, lo que hemos llamado $b_1 b_2 \cdots b_n$.

En la figura 1.7 se puede ver un ejemplo de codificación: el número -105.385. Observe que el número codificado se podría especificar más formalmente como:

$$(-1)^{signo}(1.m) \cdot 2^{exp-127}$$

donde m es la mantisa, los 23 dígitos binarios de la representación, y exp es el valor del número de 8 bits almacenado en el campo del exponente.

[2] Media, simple, doble y cuádruple precisión de 16 a 128 bits. Incluso usando base 2 o base 10.

[3] Note que se podría realizar algo similar en base 10: por ejemplo, si tenemos el número 1347, podemos expresarlo como $1.347 \cdot 10^3$, dejando el primero dígito distinto de cero como unidades.



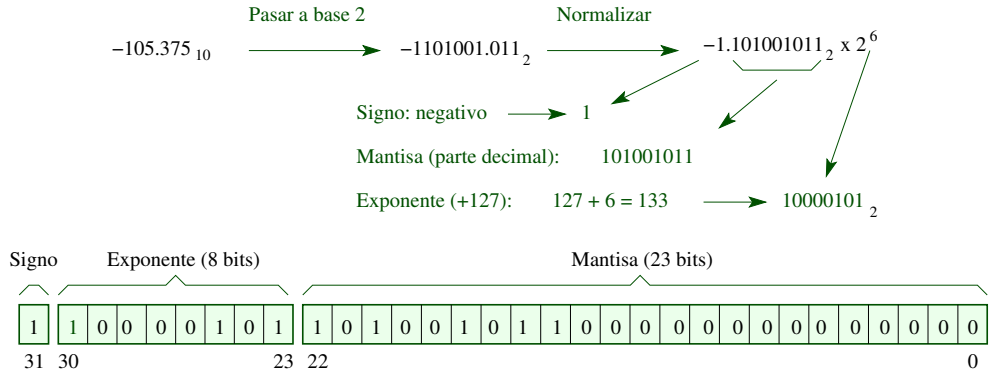


Figura 1.7
Codificar -105.375 en formato IEEE754 de simple precisión.

Seguro que ya ha considerado las limitaciones que implicará usar este tipo de representación; destacamos especialmente:

- Respecto al *exponente*. El rango de posibles exponentes está limitado. Esto implica una limitación tanto para números muy grandes (exponente muy alto, positivo) como para números muy pequeños (exponente muy bajo, negativo).
- Respecto a la *mantisa*. No podemos almacenar más de 23 dígitos binarios. Si al representar el número en binario necesitamos más de 23 dígitos, no tendremos más remedio que eliminar los menos significativos. Por tanto, los valores almacenados en muchos casos tienen que ser aproximados.

Lógicamente, una forma muy simple de contrarrestar estos problemas es aumentar el tamaño de los campos. Por ejemplo, usando datos de doble precisión (64 bits) o incluso de cuádruple precisión (128 bits).

Características adicionales

Los detalles descritos ya son suficientes para entender la idea de cómo codificar en punto flotante. Sin embargo, es interesante describir algunas características adicionales que nos indican primero la variedad de posibilidades en la codificación, y segundo algunas posibilidades más que nos ofrece el formato *IEEE754*.

En primer lugar, el rango de posibles valores del exponente se establece como $[-126, 127]$. Es decir, los números que se almacenan normalizados tienen un exponente de 8 bits que va desde el 1 hasta el 254. Los valores 0 y 255 tienen un significado especial:

- Exponente 00000000_2. En este caso el número no está normalizado. Es decir, la mantisa m no corresponde al número $1.m$, sino a un número $0.m$. De esta forma podemos almacenar un número incluso más pequeño, ya que en este caso el valor corresponde a:

  $$(-1)^{signo}(0.m)\cdot 2^{-126}]$$

  que en el caso en que la mantisa sea una secuencia de ceros con un uno al final, corresponde al valor $1\cdot 2^{-(126+23)}$, es decir, a $1\cdot 2^{-149}$. Un valor de exponente imposible de almacenar en formato normalizado.
- Exponente 11111111_2. Esta secuencia nos permite codificar valores *especiales*. En concreto podemos almacenar:
  - Valores infinitos. Si la mantisa es cero, indicamos un valor $+\infty$ (si el signo es positivo) o $-\infty$ (si el signo es negativo).

- Valores no válidos. Si la mantisa es no cero, indica que no es un número válido. Representado como `NaN` (*Not a Number*) nos sirve para representar casos especiales. Por ejemplo, la raíz de un número negativo no existe, por lo que podemos representar el resultado con `NaN`.

Finalmente, note que el valor cero se representa si el exponente y la mantisa son cero. En este caso nos encontramos dos posibilidades dependiendo del valor del bit de signo. Podemos decir que se puede representar los valores $+0$ y -0. Aunque parezca una redundancia, e incluso un inconveniente, también se puede aprovechar para mejorar la información almacenada. Por ejemplo, el valor $1/0$ es indeterminado, pero si el cero tiene signo podemos decir que es $+\infty$ o $-\infty$, según sea signo positivo o negativo respectivamente.

1.4 Arquitectura del ordenador

En el estudio de los tipos básicos, nos interesa especialmente aspectos relacionados con la *CPU* y la *memoria principal*. El diseño de la arquitectura de la computadora sugiere en gran medida los tipos de datos que se pueden usar para obtener programas eficientes. Dos conceptos básicos que debemos tener en cuenta son:

1. *Longitud de palabra*. El hardware no realiza operaciones trabajando con bits individuales, sino que se diseña para trabajar con varios bits a la vez. La longitud de palabra hace referencia al tamaño —número de bits— que se ha fijado para que el hardware trate de forma conjunta. Puede afectar a múltiples aspectos de la arquitectura: el tamaño de los datos almacenados en los registros de la CPU, el tamaño que puede operar la CPU en una tiempo mínimo, el tamaño de datos que se transfiere de un dispositivo a otro, el número de bits necesarios para especificar una dirección de memoria, etc. Por ejemplo, tamaños habituales en computadoras personales son 32 o 64 bits.
   No significa que no puedan manejarse datos de menor o mayor tamaño —de hecho, seguro que van a aparecer— pero sí que determinan un tamaño ideal para que la computadora maneje de forma eficiente. Lo habitual es que haya otros tamaños que sean fracciones o múltiplos del tamaño de palabra. Por ejemplo, podemos tener un ordenador de 32 bits que también maneja 64 bits. Es probable que las operaciones entre datos de 64 bits las realice en varios pasos, aprovechando que tiene implementadas en hardware las operaciones con 32 bits (su tamaño de palabra).
2. *Dirección de memoria*. La memoria se diseña como una tabla de secuencias de bits de cierto tamaño. Cada una de las posiciones de la tabla es una dirección de memoria distinta. En cada dirección de memoria se almacena una tupla de bits. Lo más habitual es que cada posición contenga 8 bits, es decir, un *byte*.
   En la práctica podemos pensar en la memoria como una tabla con un índice —dirección de memoria— que comienza en cero, y en la que se almacenan datos de 8 bits[4]. A nivel hardware, esa dirección de memoria no es más que un número codificado en binario: por ejemplo 32 bits.
   Lógicamente, el formato binario es muy incómodo de manejar, por lo que cuando indicamos una dirección de memoria es más fácil usar una notación en octal o hexadecimal. Como es un concepto muy cercano al hardware, cuando hagamos referencia a un ejemplo concreto de dirección de memoria usaremos la notación hexadecimal —comenzando con *0x*— para recordar que estamos usando conceptos más cercanos al hardware.
   Normalmente los datos tendrán un tamaño de más de un *byte*. Por tanto, si nuestra memoria se divide en bytes, un dato suele ocupar varias direcciones de memoria. Por ejemplo, podemos

[4]Usamos 8 porque es habitual y probablemente será el tamaño que use en su ordenador. Sin embargo, la memoria de una computadora puede diseñarse con otros tamaños.



tener un dato de 32 bits que se sitúa en la posición *0xF3A2*. En este caso, realmente tendremos ocupadas las posiciones *0xF3A2*, *0xF3A3*, *0xF3A4* y *0xF3A5*. Si situamos un dato justo a continuación, estará en la posición *0xF3A6*.

1.4.1 Datos en memoria

Un dato se almacena en una o varias posiciones de memoria. Dos aspectos importantes que debemos tener en cuenta son:

1. *Orden de los bytes*. Si ocupa varios bytes consecutivos, tenemos que determinar el orden en que se sitúan.
2. *Alineamiento*. La posición del primer byte del dato se puede alinear en determinadas posiciones de memoria.

Un dato puede tener una longitud de varios *bytes*. Por ejemplo, en un ordenador de 32 bits es muy probable que tengamos datos de esa longitud. El almacenamiento de ese dato ocupará varias posiciones de memoria. Podemos hacerlo de distintas formas. Fundamentalmente, los bytes se pueden ordenar de dos formas[5]:

- *Little-Endian*. Los bytes se ordenan desde el menos significativo al más significativo. Es el que muestra la figura 1.8.
- *Big-Endian*. Los bytes se ordenan desde el más significativo al menos significativo.

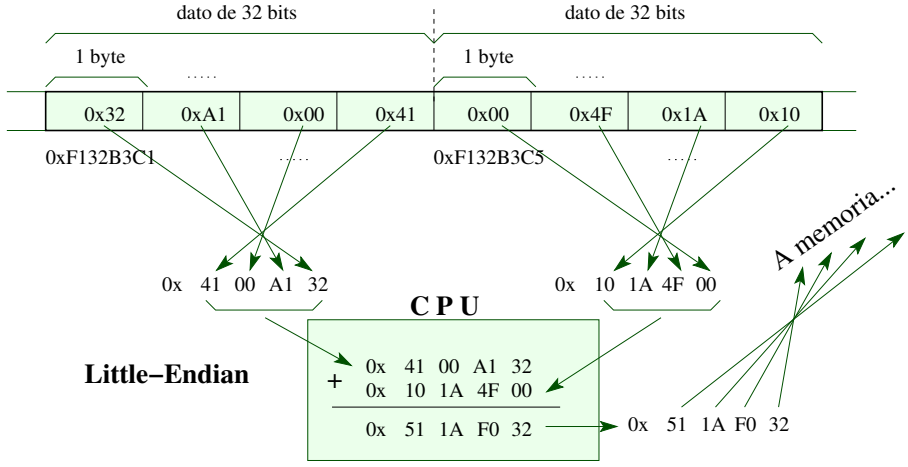


Figura 1.8
Orden de los bytes en memoria.

En la figura 1.8 se presenta gráficamente la idea de operar dos datos: *0x4100A132* y *101A4F00*, obviando detalles de cómo se realiza y el tipo de operación o datos que tenemos, podemos ver cuál es la idea de una ordenación en memoria de tipo *Little-Endian*. Observe que en todos los datos se ha añadido *0x* como prefijo para recordar que trabajamos a nivel de bit, por lo que dos dígitos corresponden a un byte.

En la memoria los datos aparecen con los 4 bytes "al revés", la operación que se realiza realmente considera los datos con los bytes en otro orden (véase la CPU en el gráfico), y el resultado obtenido vuelve a almacenarse en memoria "al revés". Note cómo nos referimos a esta representación como "al revés". Realmente esta expresión es para recordar que nuestra notación -incluso decimal- es justo la contraria: cuando escribimos un número decimal lo hacemos desde

[5]No vamos a entrar en detalles de otras posibilidades. Fundamentalmente, estas dos ordenaciones son las que se utilizan en casi cualquier ordenador

el más significativo. No significa que la CPU tenga que estar realizando cambios de orden de los *bytes*, ya que todo el sistema está diseñado para usar este orden[6].

El sistema podría estar diseñado para almacenar la información como *Big-Endian*. En este caso, los 4 bytes del dato que hay en memoria estarían en el orden inverso. Dicho de otra forma: si a un ingeniero informático le indican que en la memoria de un ordenador se almacenan los dos datos que aparecen en la figura 1.8, no podrá saber exactamente cuáles son sin conocer el orden (en inglés, *endianness*). Por ejemplo, si le dicen que es un sistema *Big-Endian*, entonces los dos datos son: *0x32A10041* y *0x004F1A10*, distintos a los que se procesan en la figura 1.8.

Por otro lado, otro aspecto importante sobre la forma en que se almacenan los datos en memoria es el *alineamiento de datos*. Por ejemplo, suponga que tenemos un dato de longitud 4 bytes. Alinearlo consiste en situarlo en ciertas posiciones de memoria que cumplan alguna condición, como que fuera múltiplo de 4.

En principio, parece lógico que pudiéramos situarlo en cualquier posición de memoria ocupando dicha posición y las 3 siguientes. Sin embargo, en una arquitectura de 32 bits de longitud de memoria, podríamos diseñar el acceso de memoria para que fuera más eficiente al acceder a las posiciones múltiplos de 4 bytes. En este caso, si queremos mejorar la eficiencia en tiempo, convendría alinearla en un múltiplo de cuatro.

En algunas arquitecturas o para ciertas operaciones podría ser incluso obligatorio el alineamiento previo para que funcionen correctamente.

En la figura 1.9 tiene un ejemplo en el que se han almacenado dos datos: uno de 16 bits y otro de 32 bits. El primero se ha almacenado en una posición múltiplo de 4. El segundo podría situarse en la posición *0xF132B3C6*, pero en lugar de eso, se ha alineado en la posición *0xF132B3C8* que también es múltiplo de 4.

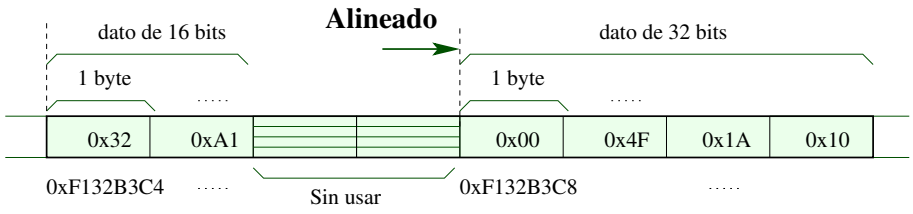


Figura 1.9
Alineamiento de datos.

Observe que en la figura 1.9 se han situado los datos de forma que se desperdician dos posiciones de memoria. Si el alineamiento es algo opcional, podríamos escoger entre optimizar la memoria o el tiempo. En este caso se ha sacrificado memoria para obtener un mejor rendimiento en tiempo.

1.5 Caracteres

En el diseño de un sistema, es importante la selección de los formatos para tipos entero y real, ya que la arquitectura se diseña para obtener los resultados más óptimos. El tipo entero y real están muy cerca del *hardware*, ya que probablemente la *CPU* implementará los algoritmos para realizar operaciones aritmético-lógicas con ellos.

El caso de representar caracteres es de alguna forma más independiente del *hardware*, siendo una codificación más lógica que física. Realmente, lo que hacemos es establecer una correspondencia entre el conjunto de caracteres que queremos representar y una secuencia de bits que los codifican.

[6]Podríamos decir que si preguntáramos a la máquina sobre nuestra forma de escritura, se extrañaría de que escribiéramos al revés.

En principio, el *hardware* no tiene que implementar ningún procesamiento especialmente para este tipo. De hecho, en C++ se puede suponer que el tipo básico **`char`** no es más que un entero de pocos bits que codifica el valor de cierta tabla de codificación: por ejemplo, la tabla *ASCII*.

El problema de la codificación de caracteres es por tanto establecer una correspondencia entre cada carácter a codificar y una secuencia de bits. El tipo de codificación puede ser:

- *De longitud fija*. Se establece un número de bits fijo. Todos y cada uno de los caracteres se les asigna una codificación de esa longitud. Por ejemplo, la tabla *ASCII* básica tiene una longitud de 7 bits.
- *De longitud variable*. No hay un número fijo de bits, sino que cambia dependiendo del carácter. Por ejemplo, en la codificación *UTF8* algunos caracteres necesitan 8 y otros 16, 24 o 32 bits.

1.5.1 Código ASCII: Extensiones

El código *ASCII* propiamente dicho corresponde a una tabla de codificación de 128 posiciones, ya que se usan sólo 7 bits. Se empieza a diseñar hace más de medio siglo, y se enfoca a la codificación de textos que usan el alfabeto inglés. La tabla corresponde a la mitad superior de la figura 1.10. Observe que se incluyen tanto caracteres imprimibles como no imprimibles. Cualquier texto escrito en inglés se puede representar como una secuencia de caracteres de esta tabla, es decir, se puede guardar como una secuencia de tuplas de 7 bits.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8x	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9x	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGC	SCI	CSI	ST	OSC	PM	APC
Ax	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Figura 1.10
Tabla de codificación ISO-8859-1.

Lógicamente, para otros lenguajes no es suficiente. Por ejemplo, en español necesitamos las tildes, la diéresis y la ñ para poder representar cualquier texto. En otros, además, podrían aparecer otros caracteres que necesiten de otros signos diacríticos. Para ampliar las posibilidades de representación se crea lo que denominamos un código *ASCII extendido* añadiendo un bit, es decir, un código de 8 bits (por tanto, una tabla de 256 posibilidades) en el que la primera mitad corresponde al código *ASCII* y la segunda a la extensión. En la figura 1.10 se presenta el estándar ISO del alfabeto latino, con el que es posible escribir cualquier texto de un conjunto de lenguas de Europa occidental.

La codificación ISO-8859-1 nos permite escribir cualquier texto almacenando cada carácter en un único objeto **`char`**. En este caso, el tipo **`char`** puede guardar únicamente un *byte* que

corresponde al código *ASCII* correspondiente o al código extendido si el carácter está en la segunda mitad de la tabla. Por simplicidad, usaremos el término *tabla ASCII* para referirse a una codificación extendida, aunque realmente corresponda sólo a los primeros 128 caracteres.

Realmente no es la única codificación posible y de hecho se han creado otras tablas que se han usado ampliamente, aunque en muchas de las codificaciones se ha respetado que las primeras 128 entradas corresponden a la tabla *ASCII* de forma que sea compatible con la codificación inglesa original. Por ejemplo, otro estándar ISO es el 8859-15 (véase la tabla 1.10 en la página 13) que corresponde a una pequeña modificación del 8859-1 con la intención de cambiar algunos caracteres, entre ellos, el carácter del euro.

Otra codificación de 8 bits ampliamente usada es la conocida como *windows-1252* (o *CP1252*). Diseñada por *Microsoft* para su sistema operativo y que también se basa en una ampliación del código *ASCII* para el alfabeto latino. La mayor parte de los caracteres del *ISO-8859-1* están situados en la misma posición de *windows-1252*. Incluye el carácter del euro, así como nuevos caracteres imprimibles.

Note que el lenguaje C++ está definido sobre el código *ASCII* básico. No podemos definir ningún identificador que use caracteres fuera de la tabla *ASCII*. Por ejemplo, no es posible incluir el carácter `'ñ'` en el nombre de una variable. Lógicamente, seguro que ha usado cadenas de caracteres o texto que incluye otros caracteres, pero recuerde que eso son datos que no forman parte del lenguaje.

1.5.2 Unicode

El problema de la adaptación del software a distintos idiomas, la transmisión y visualización de textos de otros idiomas, o incluso la posibilidad de usar varios idiomas en un mismo sistema hace que la solución que hemos indicado en las secciones anteriores no se pueda usar. Para resolver este problema se crea el estándar *Unicode*, donde se incluyen todos los caracteres asociando a cada uno de ellos un código único.

El estándar ha ido creciendo, añadiendo nuevos caracteres hasta alcanzar decenas de miles. Aunque inicialmente un valor de 16 bits parecía más que suficiente para codificar todos los caracteres actuales, pronto se amplió el estándar para poder usar 32 bits, pudiendo alcanzar millones de entradas. Ahora bien, una vez determinados los caracteres y asignados los códigos correspondientes, será necesario seleccionar una forma de tratarlos en el ordenador.

La solución más simple para realizar una codificación es mediante longitud fija, por lo que podemos establecer una correspondencia entre cada carácter *Unicode* y una secuencia de 32 bits. Sin embargo, no siempre es la mejor solución. Por ejemplo, si usamos una codificación de 32 bits, cada carácter que queramos almacenar necesitará 4 bytes. Eso ocurre tanto para una letra común como alguna del alfabeto inglés, como para un carácter de una determinada lengua muy local que incluso puede que no se esté usando.

Podemos mejorar la eficacia de nuestra codificación si aplicamos una longitud variable. En este caso, los caracteres más habituales se codifican con menos bits. Por ejemplo, las letras del alfabeto inglés se pueden codificar con un único byte, mientras que un carácter de una lengua muerta se codifica con 4 bytes. Lo importante es que no se puedan confundir las codificaciones, es decir, que se pueda codificar y decodificar de forma única.

UTF

Dado que hay distintas formas de codificación, *Unicode* define distintas alternativas para que podamos seleccionar una de ellas como la codificación estándar de nuestro *software*, de forma que sea compatible con cualquier otro que la use. Usando el acrónimo *UTF* (**U***nicode* **T***ransformation* **F***ormat*) se establecen distintas formas de codificación. Podemos destacar las siguientes:



- *UTF32*. Una codificación fija de 32 bits. Esta resulta simple e intuitiva, ya que si el estándar Unicode usa 32 bits para clasificar todos los caracteres, lo más sencillo es crear una codificación de la misma longitud para que un ordenador los procese.
- *UTF16*. Una codificación de longitud variable de 16 bits mejora la eficacia de la codificación. Tenga en cuenta que el número de caracteres que resuelven la mayoría de los caracteres habitualmente usados en el mundo se pueden codificar con menos de 16 bits. Si queremos evitar un gasto excesivo podemos codificar cada uno de ellos con un grupo único de 16 bits, y dejar los caracteres "menos habituales" codificados con dos grupos de 16 bits.
- *UTF8*. Una codificación de longitud variable de 8 bits permite tener caracteres codificados con un único byte. Lógicamente, un carácter cualquiera puede codificarse con 1, 2, 3, o 4 bytes.

La codificación *UTF16* es un buen compromiso entre tener un tamaño fijo en la codificación y no desperdiciar mucho espacio para almacenar un carácter. Podríamos incluso crear sistemas que trabajaran sólo con el subconjunto de caracteres codificados con un único grupo de 16 bits[7], que serían válidos prácticamente en cualquier lugar del mundo. Esta codificación es la que usa *Microsoft Windows*.

La codificación *UTF8* está especialmente diseñada para ser compatible con la tabla *ASCII*. Dado que es posible asignar códigos de 1 byte, podemos asignar los mismos códigos para la primera mitad de la tabla *ASCII*. Por ejemplo, cualquier texto inglés codificado en *ASCII* es idéntico en *UTF8*. Para otros caracteres, como por ejemplo los que hemos añadido en la parte extendida para Europa occidental, se podrán codificar con más de un byte. Lógicamente, los más habituales requieren solamente dos bytes. La ventaja de esta codificación es que gran parte de los documentos y software que se ha creado para trabajar con el código *ASCII* sigue siendo válido prácticamente sin modificación. Este sistema es ampliamente usado, por ejemplo, en los sistemas *GNU/Linux*.

Endianness y UTF

La codificación en *UTF* es un caso en el que es necesario especificar el orden de los bytes (es decir, la *endianness*). Por ejemplo, si tenemos una codificación *UTF32* cada carácter codificado ocupará 4 bytes. Estos bytes se podrían almacenar de dos formas: desde el byte más significativo (*big-endian*) o desde el menos significativo (*little-endian*). Algo similar ocurre para el caso de *UTF16*, donde tendremos que escoger la ordenación de los dos bytes que componen los bloques de 16 bits.

La codificación *UTF* permite utilizar cualquiera de las dos *endianness*. Por lo tanto, podemos decir que en lugar de tener tres codificaciones —*UTF8, UTF16, UTF32*— tenemos 5 codificaciones: *UTF8, UTF16LE, UTF16BE, UTF32LE, UTF32BE*.

Dado que tenemos distintas posibilidades, es importante conocer exactamente la que estamos usando. Para poder realizarlo se puede añadir como primer carácter del fichero un carácter especial que determina la codificación. Para ello, lo único que se hace es añadir el carácter con código *U+FEFF* como primer carácter. Este es un carácter que se considera nulo, pero determina exactamente cómo están guardados los datos en el archivo. En concreto, los valores de los primeros bytes se presentan en la tabla 1.2, junto con sus significado.

Observe que hemos añadido el caso de *UTF8*. En realidad no es necesario especificar el orden para esta codificación, ya que no existe ninguna ambigüedad: la secuencia de bytes para codificar en UTF8 es siempre la misma. Sin embargo, podemos añadir el carácter *U+FEFF* al principio del archivo. Lógicamente, este carácter es nulo y podemos ignorarlo, pero nos sirve para reconocer que el resto del archivo se debería interpretar con esta codificación. Como puede ver, el carácter *U+FEFF* se codifica en *UTF8* con 3 bytes.

[7] Lo que se denomina el *plano básico multilingüe*.

Tabla 1.2
Carácter BOM y endianness.

Bytes iniciales	Codificación correspondiente	Endianness
00 00 FE FF	UTF-32	big-endian
FF FE 00 00	UTF-32	little-endian
FE FF	UTF-16	big-endian
FF FE	UTF-16	little-endian
EF BB BF	UTF-8	no aplicable

Finalmente, el carácter *BOM* es algo opcional, no es necesario añadirlo al principio del archivo. Esto será posible si sabemos la codificación que estamos usando. Normalmente, si se conoce la codificación de un archivo, se recomienda no usar el carácter *BOM*.

En otros casos, puede ser conveniente, si no necesario, evitar la marca *BOM*. Por ejemplo, si el sistema *GNU/Linux* se codifica por defecto en *UTF8*, puede resultar conveniente evitar las marcas. Es el caso de los archivos de texto que contienen *shell scripts*, que se supone que siendo de texto comienzan con los caracteres `"#!"`.

Finalmente, es interesante que tenga en cuenta la posibilidad de estos bytes iniciales cuando usa un editor de texto. En algunos casos el editor podría no presentar nada, pero sí almacenar esos valores para indicar la codificación. Si es así, tal vez otro programa lea el archivo y falle porque no los reconoce. También es posible que use un editor que refleja el contenido de esos bytes como primeros caracteres del archivo. Recuerde que si usa un editor de código *ASCII* y carga uno de estos ficheros, encontrará "caracteres extraños" al principio.

En nuestros programas ignoraremos esta posibilidad: los programas se implementarán generalmente pensando en una codificación fija como el código *ASCII extendido*. Muchos de ellos seguirán siendo correctos para trabajar con *UTF8* que no incluye ninguna marca *BOM*. Algunos podrán fallar, ya que la lectura de un carácter —un **`char`**— puede corresponder a la lectura de "medio código". Por ejemplo, una letra acentuada en *UTF8* ocupará dos bytes, y serán necesarios dos objetos de tipo **`char`** para almacenarla.

1.6 Problemas

Problema 1.1 Transforme el número -347 a base 2 considerando la representación en complemento a 2 con 16 bits.

Problema 1.2 Transforme el número $347.5/1024$ a base 2 considerando la representación *IEEE754* de simple precisión (32 bits).

Problema 1.3 Considere el número -23765 representado en complemento a 2 con 32 bits. Suponga que se almacena en una posición de memoria. Después de almacenarlo, suponga que accedemos a esa misma posición de memoria pero para consultar el dato **en coma flotante IEEE754 de simple precisión** que se almacena. ¿Qué valor obtendría?



2 Tipos básicos: Bits

Introducción 17
Tipos aritméticos 18
Características de los tipos 22
Tipo enumerado 24
Conversiones 26
Tipos de datos y bits 31
Problemas 39

2.1 Introducción

Los tipos simples **`bool, char, int, double`** son suficientes para la mayoría de las aplicaciones. Con ellos podemos representar objetos simples que incluyan información básica, o usar los tipos compuestos —**`vector, string`** y **`struct`**— en base a ellos para crear estructuras más complejas.

Sin embargo, el desarrollo de aplicaciones de mayor tamaño, incluyendo mejores soluciones en tiempo y espacio, requieren un estudio más profundo de las posibilidades que nos ofrece el lenguaje. Un conocimiento preciso de los tipos más básicos del lenguaje nos permite seleccionar la cantidad de memoria adecuada, la precisión, o el rango de posibles valores de cada dato.

En este tema se estudiarán los tipos más básicos. Serán la base sobre la que iremos desarrollando nuevas posibilidades en los siguientes temas, con estructuras de datos más complejas, a un nivel de abstracción superior que nos ofrecerá herramientas cada vez más potentes para, en última instancia, diseñar fácilmente soluciones a problemas complejos. Para introducirnos en esta *metodología*, comencemos estudiando los elementos más básicos.

2.1.1 Tipos y lenguaje de alto nivel

Los lenguajes de alto nivel nos permiten desarrollar software independientemente de los detalles del hardware donde se ejecuta. En nuestro caso, los programas C++ que desarrollamos en este curso se pueden ejecutar sobre una amplia gama de ordenadores, incluyendo máquinas de distinta longitud de palabra, distinto orden de bytes en memoria (*indianness*), distinto sistema operativo, etc. Sin embargo, es importante conocer los detalles de la representación para:

1. Desarrollar software de bajo nivel. Por ejemplo, algún módulo para interaccionar directamente con un dispositivo, o un módulo al nivel del sistema operativo.
2. Entender mejor las limitaciones y posibilidades que nos ofrece el lenguaje. Aunque desarrollemos una aplicación a alto nivel, los tipos de datos tienen unas características que limitan sus posibilidades.

3. Comprender el comportamiento del software y cómo relacionarlo con posibles errores. Si el software falla, conocer los detalles de cómo funciona nos puede ayudar a entender mejor qué está haciendo el programa y por tanto cómo podría resolverse.

En general, nuestras aplicaciones podrán implementarse sin conocer detalles de la representación. Sólo será necesario entender algunas ideas básicas, como los tamaños relativos de los tipos, el uso o no de signo, las limitaciones en la precisión, etc. y nuestra aplicación será válida para casi cualquier sistema. Por ejemplo, si una arquitectura requiere que los datos enteros se alineen en ciertas posiciones de memoria, el compilador buscará las posiciones adecuadas. Incluso si una arquitectura permite los datos alineados o no —con efectos en el tiempo de ejecución— el compilador puede incluir opciones de alinear para obtener mejor tiempo o no alinear para obtener mejor aprovechamiento de memoria.

El lenguaje C++ no fija los detalles de cómo es cada uno de los tipos que maneja. En lugar de eso, establece una serie de condiciones para que funcionen de manera lógica, y ofrece una serie de herramientas básicas para que nuestros programas puedan consultar y tener en cuenta sus detalles. En otros lenguajes puede encontrar exactamente en qué consiste cada tipo de dato. En C++ cada compilador suele adaptar los tipos básicos al sistema sobre el que se trabaja de forma que el resultado de la compilación resultará más eficiente.

2.2 Tipos aritméticos

Al conjunto de tipos que representan valores enteros y reales se denomina tipos *aritméticos*. Más exactamente, se clasifican como:

1. Tipos *integrales*. Estos incluyen a su vez:
   a) Tipo *booleano*.
   b) Tipos *carácter*.
   c) Tipos *enteros*.
2. Tipos en *coma flotante*.

2.2.1 Tipo booleano

El tipo **`bool`** en C++ nos permite almacenar un valor booleano. Un objeto de este tipo puede tomar uno de entre dos posibles valores: **`true`**, **`false`**. Recordemos que el tipo **`bool`** no existe en C, sino que se utilizan enteros en su lugar, representando el valor **`false`** con el número 0 y el valor **`true`** con un valor distinto de cero (por ejemplo, 1).

Aunque en C++ se utiliza un tipo específico, no podemos ignorar la forma en que se resuelve en C, ya que el código de este lenguaje también es válido en C++. Además, no sólo tenemos varias formas de representar un valor booleano, sino que también se pueden mezclar en expresiones que tienen sentido mediante conversiones.

Es posible que incluso haya relacionado los valores de tipo **`bool`** con los enteros, ya que si escribe un booleano en **`cout`**, puede obtener un valor 0 o 1; en caso de ser **`false`** o **`true`**, respectivamente. También en la lectura de un booleano se usará esta representación numérica[1].

En general, recuerde que cualquier tipo integral —y más adelante veremos que también para otros tipos— se puede usar en el lugar de una expresión de tipo booleano. Por ejemplo, puede ser el valor usado como condición en una sentencia condicional, en un bucle, como operando de una expresión booleana, etc. En todos los casos, la clave será preguntarse si el valor es cero o distinto cero; en el primer caso corresponderá al valor **`false`** y en el segundo al valor **`true`**.

[1] Realmente, no tiene que ser así, ya que si activamos el `flag boolalpha` en un flujo de entrada o salida, se pueden usar las representaciones de texto **`true`**, **`false`**. Estas dos cadenas de texto podrían modificarse para adaptarse al lenguaje usando una *localidad* (**`locale`**) adecuada.



2.2.2 Tipos carácter

El lector seguro que conoce el tipo **char**. En la mayoría de los casos lo usaremos de forma simplificada al considerar que el texto no es más que caracteres alfabéticos contiguos que almacenan un código ASCII. Además, es interesante recordar que además de este tipo el lenguaje ofrece el tipo **wchar_t**, que permite almacenar juegos de caracteres más grandes (como *Unicode*). Todo esto en el contexto del estándar C++98, ya que en el estándar C++11 aparecen nuevos tipos carácter.

Una cuestión interesante se refiere al rango de valores de un objeto de tipo **char**. Generalmente tiene un tamaño de 8 bits, aunque el rango no está determinado de forma precisa. En algunos casos podemos considerar $[-128, 127]$ y en otros $[0, 255]$. Depende del compilador.

Si se desea utilizar un determinado rango de valores, podemos optar por los tipos **signed char** (con signo) y **unsigned char** (sin signo). Aunque hay que tener en cuenta que, junto con **char**, constituyen 3 tipos distintos (a pesar de ello, **char** se comportará como uno de los otros dos, ya que tiene el mismo rango).

Tal vez se pregunte si se pueden realizar las mismas operaciones con este tipo —de sólo un byte— como con un tipo entero. El tipo **unsigned char**, como cualquier integral, es compatible con las mismas operaciones aritméticas y lógicas que el tipo **int** (véase sección 2.6.1).

Por ejemplo, imagine que tenemos una aplicación de tratamiento de vídeo con imágenes en blanco y negro, donde un cuadro —una imagen— tiene miles de puntos que corresponden a un nivel de gris (desde el negro, hasta el blanco). ¿Cómo podemos representar la información de una imagen?

Una imagen se puede considerar una matriz, por ejemplo, de 768 columnas y 576 filas. Por lo tanto, contiene 442368 —768×576— puntos distintos; cada uno de ellos almacenando un valor de gris. Podemos usar un rango de 256 posibles valores, donde el valor 0 corresponde al color negro y el valor 255 al color blanco (por ejemplo, el 128 sería un gris medio, el 25 un gris oscuro, etc.).

En esta aplicación es importante el tamaño de un punto, ya que podemos mantener en memoria millones de ellos. Podemos declarar un valor de gris como un tipo **unsigned char**. Con ello, sólo necesitamos un byte para cada punto de la imagen. Es decir, 432 Kbytes para cada imagen.

2.2.3 Tipos enteros

El tipo **int** es uno de los más habituales. La mayoría de los casos en los que necesitamos un valor entero, lo resolvemos con este tipo, aunque realmente tenemos distintas alternativas para representar enteros. Así, podemos diferenciar entre:

- Tipo **int**. Corresponde a enteros con signo. También se puede nombrar **signed int**, o incluso **signed**.
- Tipo **unsigned int**. Enteros sin signo. También se puede nombrar **unsigned**.

Por ejemplo, en la máquina sobre la que normalmente trabajo, **int** está en el rango $[-2147483648, 214748$ mientras que **unsigned int** está en el rango $[0, 4294967295]$. Note que en este caso no es necesario representar negativos, y por tanto tenemos todos los bits disponibles para la magnitud.

Además, podemos escoger entre tres tamaños:

- Tipo **short int**. Corresponde a entero corto. También se puede nombrar **short**.
- Tipo **int**.
- Tipo **long int**. Corresponde a entero largo. También se puede nombrar **long**.

Por ejemplo, en mi máquina, **short int** está en el rango $[-32768, 32767]$ (usa dos bytes para la representación), mientras que el tipo **unsigned short int** está en el rango $[0, 65535]$.

A pesar de sus nombres, con palabras como **short** o **long**, no está garantizado que estos tres tipos sean de distinto tamaño (véase la sección 2.3.1). Por ejemplo, en una de mis máquinas, **int** y **long int** tienen el mismo tamaño —4 bytes— aunque en otra máquina es distinto: 4 y 8 bytes, respectivamente.

Recordemos que se puede escribir un literal entero de la forma que habitualmente escribimos los enteros, es decir, un signo seguido por una serie de dígitos decimales. Por ejemplo, podemos escribir dos dígitos como *17*. En este caso, es importante saber que el compilador asume que es de tipo **int**[2].

Si queremos indicar un literal que corresponde a otro tipo que no sea **int**, podemos hacerlo explícitamente con el sufijo *U* para indicar que corresponde a valores **unsigned**, o el sufijo *L* para indicar que es **long**. Por ejemplo, podemos escribir `17L` como literal de tipo **long int**, o `17UL` para indicar **unsigned long int**.

Por otro lado, se pueden escribir enteros en notación octal y hexadecimal. Para ello, debemos preceder el número con un carácter `'0'` (para expresar octal) o con los caracteres `'0x'` (para expresar hexadecimal).

2.2.4 Tipos en coma flotante

El estándar de C++ distingue tres tamaños para representar un número en coma flotante:

- Tipo **float**. Precisión simple.
- Tipo **double**. Precisión doble.
- Tipo **long double**. Precisión extendida.

aunque como el caso de los enteros, no está garantizado que tengan distintos tamaños(véase la sección 2.3.1). Por ejemplo, en mi máquina, calculando los valores del mínimo, máximo y épsilon obtengo:

- Tipo **float**.
  - Mínimo: 1.17549e-38.
  - Máximo: 3.40282e+38.
  - Épsilon: 1.19209e-07.
- Tipo **double**.
  - Mínimo: 2.22507e-308.
  - Máximo: 1.79769e+308.
  - Épsilon: 2.22045e-16.
- Tipo **long double**.
  - Mínimo: 3.3621e-4932.
  - Máximo: 1.18973e+4932.
  - Épsilon: 1.0842e-19.

donde *épsilon* —ε— corresponde a la diferencia entre el valor *1.0* y el menor valor mayor que *1.0*. Es decir, no es posible representar ningún número del intervalo $(1, 1+\varepsilon)$. Con este valor nos podemos hacer una idea de cuál es la precisión con la que representamos los números. Por otro lado, observe que el valor mínimo corresponde a un número muy pequeño, muy cercano al cero. Lógicamente, si recuerda la representación *IEEE 754* (sección 1.3.2) cualquier número positivo también se puede representar como negativo.

La forma habitual que usamos para escribir un número en coma flotante corresponde al tipo **double**. Por ejemplo, si escribimos `1.4142`, el compilador lo considera un literal de tipo **double**. Podemos usar los sufijos *F* para indicar que es **float** y *L* para indicar que es **long double**.

Finalmente, es interesante recordar (véase sección 1.3.2) que en el caso de los números en coma flotante, es probable que la representación también nos permita almacenar valores especiales como infinito —positivo o negativo— o incluso `NaN` para indicar que el objeto no contiene un número. Por ejemplo, si pruebo el siguiente código en mi máquina:

[2] Realmente depende de su valor, por ejemplo, si es demasiado grande le hará corresponder un **long int**, y si tampoco es representable en este tipo, **unsigned long int**.



```
cout << "Raíz de -1 vale " << sqrt(-1.0) << endl;
cout << "0.0/0.0 vale " << 0.0/0.0 << endl;
cout << "1e1000 vale " << 1e1000 << endl;
cout << "-1e1000 vale " << -1e1000 << endl;
```

obtengo como resultado:

Consola

```
Raíz de -1 vale -nan
0.0/0.0 vale -nan
1e1000 vale inf
-1e1000 vale -inf
```

donde podemos ver que el programa:

- Asigna un valor *NaN* a la raíz de -1 y a la división indefinida de $0.0/0.0$. Este resultado es lógico, puesto que no existen los valores reales que corresponden a esos cálculos.
- Escribe *inf* cuando queremos usar el literal `1e1000`. Es una forma sencilla y eficaz para resolver el problema de que ese número no es representable como **`double`** en mi sistema. Como es un número mayor que el máximo, el compilador le asigna infinito.

2.2.5 Sinónimos con typedef

Los rangos y precisiones de los tipos que hemos visto dependen del sistema donde se compile el programa y no del lenguaje. Por lo tanto, no podemos estar seguros de las características de nuestros tipos de datos. En la mayoría de las aplicaciones es un problema menor, ya que los tamaños mínimos que nos garantiza el lenguaje son suficientes para saber que nuestros programas funcionarán sin problemas. Pero, ¿y si queremos implementar una aplicación independiente del sistema que la ejecutará?

Una forma muy eficaz de garantizar que un código funciona exactamente igual en distintos sistemas es crear tipos de datos que tengan exactamente el tamaño que nosotros necesitamos. Por ejemplo, imagine que para resolver un problema necesitamos un entero con signo de 32 bits. Lo ideal sería que el lenguaje nos ofreciera un tipo con un nombre como *Int32* que tuviera estas características. El problema es que el lenguaje no lo incluye.

Para resolverlo, podemos usar **`typedef`**. Esta palabra reservada nos permite crear sinónimos de cualquier tipo de dato, ya sea básico o no. En nuestro caso, donde queremos tener un tipo *Int32* podemos escribir:

```
typedef int Int32;
```

Después de esta línea, ya disponemos del nuevo nombre. Este nuevo nombre es idéntico al tipo **`int`**. Si declaramos tres variables con este nombre:

```
Int32 a,b,c;
```

será equivalente a haberlas declarado como:

```
int a,b,c;
```

Es probable que esté pensando que esto no nos resuelve nada. Ciertamente, el tipo **`int`** no tiene por qué tener 32 bits. Por ejemplo, es posible que una máquina tenga un tamaño de entero de 16 bits, mientras que el entero de 32 bits está representado con el tipo **`long int`**. En esa máquina, el sinónimo que hemos creado no es válido. Para resolverlo, tendríamos que modificarlo:

```
typedef long int Int32;
```

Esta solución implica una mejora, ya que para cambiar de máquina sólo tengo que modificar una línea para que todo el código sea válido. Aunque no es la mejor solución, puesto que lo óptimo es que el código sea válido en cualquier máquina sin modificar nada. Para resolver esto, lo mejor sería incluir las dos líneas y que el compilador pudiera seleccionar la que más le conviene. Este código es posible (véase por ejemplo, las directivas condicionales del precompilador) aunque queda fuera de los objetivos de este tema.

Lo fundamental en este tema es que sea consciente de que el lenguaje permite crear sinónimos de tipos de datos. Puede crear o encontrar código C++ que no contenga los identificadores que hemos presentado anteriormente, sino otros nuevos[3] que se han creado a medida con **`typedef`**.

Esta metodología ya se usa directamente en el lenguaje C++ para ofrecer algunos sinónimos de tipos para problemas concretos. Por ejemplo, en el fichero cabecera **`cstddef`** se incluye:

- Tipo *`size_t`*. Corresponde a un tipo sin signo, y se usa para indicar un tamaño. Por ejemplo, el número de elementos contenidos en un vector o el número de caracteres que componen un texto. En estos casos, el número corresponde a un valor mayor o igual que cero. Una función que lo usa, por ejemplo, es **`strlen`** incluida en **`cstring`**. Esta función devuelve el tamaño de una cadena de caracteres. El tipo de objeto que devuelve es *`size_t`*.
- Tipo *`ptrdiff_t`*. Corresponde a un tipo con signo, y representa diferencias entre direcciones de memoria[4]. Tenga en cuenta que la diferencia entre dos direcciones de memoria puede ser negativa si la primera es mayor. En este caso, la diferencia corresponde a un valor entero con signo.

2.3 Características de los tipos

El rango, precisión y características de los tipos que hemos expuesto dependen de la plataforma en la que se desarrolle. Las únicas garantías que el lenguaje ofrece son los tamaños mínimos que presenta la tabla 2.1.

Tabla 2.1
Tamaño mínimo de los tipos básicos.

Tipo	Tamaño mínimo
char	1
short int	2
int	2
long int	4
float	4
double	8
long double	8

Por lo tanto, para realizar un programa independiente de la plataforma, tendremos que conocer estas limitaciones. No es posible desarrollar un programa contando con una capacidad que esté fuera de esta tabla. Por ejemplo, si utilizo un objeto de tipo **`unsigned int`** para almacenar la población de una ciudad, no podré garantizar que el programa funciona en cualquier máquina porque tal vez el tamaño del objeto sólo me permite almacenar hasta 65536.

[3] De hecho, el lenguaje C en su estándar C99 ya incluye algunos tipos con nombres similares para garantizar un tamaño concreto.

[4] Realmente es una diferencia entre punteros, por lo que no tiene que representar el número de bytes en que difieren dos direcciones de memoria



Lógicamente, esta consideración tiene sentido si queremos crear un programas que garantiza la máxima portabilidad. Por ejemplo, en nuestro caso los programas que creamos se ejecutarán en ordenadores de escritorio o portátiles con longitud de palabra de 64 bits, lo que casi garantiza que el tipo entero va a tener al menos 4 bytes. En nuestros ejemplos asumimos ese tamaño mínimo para el tipo **int**.

Finalmente, es importante recordar que el tamaño de un tipo es el mismo que si tiene un modificador de signo. Por ejemplo, el tamaño de **int** es idéntico al tamaño de **unsigned int**. En la tabla 2.1 se ha presentado una sóla entrada por tipo independiente de los signos. Lógicamente, podríamos haber listado todos los posibles tipos:

```
1 signed char, char, unsigned char,
2 signed short int, unsigned short int,
3 signed int, unsigned int,
4 signed long int, unsigned long int,
5 float, double, long double
```

Observe que en el caso de los **int** (líneas 2, 3 y 4) el no incluir modificador es equivalente a añadir **signed**. No es igual para los **char** (línea 1) donde el tipo **char** no se sabe si es con signo o sin él.

2.3.1 Tamaños. El operador *sizeof*

El lenguaje C++ incluye el operador **sizeof**, que devuelve el tamaño de un objeto o un tipo. Tiene un único operando: un objeto o un identificador de tipo; devuelve como resultado su tamaño. En C++, el tamaño de un **char** es uno, y si asumimos que un **char** ocupa un byte, podemos considerar que **sizeof** devuelve el número de bytes que ocupada un objeto o tipo.

Por ejemplo, en mi máquina, **sizeof**(**short int**) es 2, mientras que **sizeof**(**int**) es 4. El único valor que se conoce es **sizeof**(**char**) que vale uno[5]. El lenguaje sólo garantiza:

```
sizeof(char) <= sizeof(short) <= sizeof(int)<= sizeof(long)
sizeof(char) <= sizeof(bool) <= sizeof(long)
sizeof(char) <= sizeof(wchar_t) <= sizeof(long)
sizeof(float) <= sizeof(double) <= sizeof(long double)
```

sin olvidar que los tipos con signo y sin signo tienen el mismo tamaño.

Además, el operador **sizeof** se puede aplicar a cualquier tipo, incluyendo los definidos por el usuario.

Ejercicio 2.1 Escriba un programa para presentar los tamaños de los tipos mostrados en la tabla 2.1.

2.3.2 Límites numéricos

Si se desea tener en cuenta los límites exactos que presentan estos tipos, la biblioteca estándar de C++ ofrece algunas utilidades. En primer lugar, podemos acceder a las constantes que define C para alguno de estos límites. Así, el archivo **climits** incluye algunas constantes para los tipos enteros, y el archivo **cfloat** para tipos flotantes. Algunos ejemplos son *INT_MIN* (mínimo entero) y *FLT_MAX* (máximo flotante).

En C++, se incluyen estas constantes y se amplían las posibilidades incluyendo nueva información con el uso de la plantilla **numeric_limits**. Aunque no vamos a entrar en mucho detalle (véase por ejemplo, Josuttis[17], página 115 y siguientes), el formato general para acceder a algún dato sobre estos tipos es:

[5] No significa que sea un byte. Todos los tamaños que indica **sizeof** son múltiplos del tamaño de un **char**. Aunque si un **char** ocupa un byte, podemos pensar en bytes.

numeric_limits<TIPO>::MIEMBRO

donde especificamos el tipo *TIPO* del que queremos consultar la "propiedad" *MIEMBRO*. Por ejemplo, si deseamos conocer los límites del tipo **int**, podemos escribir:

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

int main()
{
   cout << "Mínimo:" << numeric_limits<int>::min() <<endl;
   cout << "Máximo:" << numeric_limits<int>::max() <<endl;
}
```

Si deseamos conocer los límites del tipo **double**, el número de dígitos decimales, y épsilon (la diferencia entre 1 y el menor valor mayor que uno):

```
cout << "Mínimo:" << numeric_limits<double>::min() <<endl;
cout << "Máximo:" << numeric_limits<double>::max() <<endl;
cout << "Dígitos dec.:" << numeric_limits<double>::digits10 << endl;
cout << "Épsilon:" <<  numeric_limits<double>::epsilon() << endl;
```

Otras utilidades interesantes que se incluyen para los tipos en coma flotante se refieren a los valores especiales de *infinito* y *NaN*. Por ejemplo, con las siguientes líneas:

```
double n1 = numeric_limits<double>::quiet_NaN();
double n2 = -numeric_limits<double>::infinity();
```

almacenamos los valores *NaN* y $-\infty$ en las variables *n1* y *n2*, respectivamente.

2.4 Tipo enumerado

El tipo enumerado es una herramienta para generar nuevos tipos de datos definidos por el usuario. Aunque existen métodos más potentes para definir nuevos tipos, los enumerados constituyen una forma muy simple de resolver el problema de crear un nuevo tipo cuyo dominio sea un conjunto de valores pequeño y finito. Considere problemas como:

- Un juego de cartas en el que una carta viene determinada por un par de valores *Número/Palo*. El palo de la carta puede ser cualquiera de los siguientes 4 valores: *oros, copas, espadas* y *bastos*.
- Una aplicación que maneja fechas. Podemos almacenar un objeto con el valor del día de la semana. Las posibilidades son siete: *lunes, martes, miércoles, jueves, viernes, sábado* y *domingo*.

En los dos casos, tenemos un dato que puede tener un valor de entre un conjunto pequeño y finito. Una posible solución sería codificar ese dato con algún otro tipo básico de los que hemos visto. Por ejemplo, si quiero almacenar el palo de una carta podría usar:

- Un entero (**int**); Podemos establecer 4 correspondencias: 0 corresponde a *oros*, 1 a *copas*, 2 a *espadas* y 3 a *bastos*.
- Un carácter (**char**); En este caso usamos objetos de tipo **char**. Por ejemplo, 'o' corresponde a *oros*, 'c' a *copas*, 'e' a *espadas* y 'b' a *bastos*.

Si deseamos escribir un código más legible, sería más sencillo crear un nombre a ese tipo de dato y además añadir un identificador para cada uno de los posibles valores. Es una solución similar al tipo booleano, donde se ha creado un nombre —**bool**— y dos valores concretos —true, false— para el tipo. Por ejemplo, para el caso de las cartas podemos crear un tipo *Palo* que puede tomar 4 valores: *Oros, Copas, Espadas* y *Bastos*. Para realizarlo en C++98 usamos la palabra reservada **enum** con el siguiente formato[6]:

[6] En el estándar C++11 hay un nuevo tipo de enumerado que probablemente sea más adecuado.



enum <nombre tipo> { <valores> };

donde *<nombre tipo>* es el identificador del nuevo tipo y los valores son un conjunto de identificadores separados por comas. Para nuestro ejemplo de representar un *palo*, podemos hacer:

```
enum Palo {Oros, Copas, Espadas, Bastos};
```

que nos permite disponer del nuevo tipo `Palo` con 4 posibles valores: `Oros`, `Copas`, `Espadas` y `Bastos`.

El conjunto de operaciones que se pueden realizar con un tipo enumerado es bastante limitado, ya que la interpretación depende de la aplicación. En principio, las operaciones básicas son la de asignación y comparación. Por ejemplo, podemos hacer:

```
Palo p;

p= Oros;
if (p==Oros)
   cout << "El palo es Oros" << endl;
```

La entrada y salida no funciona como podría esperarse. Por ejemplo, si escribimos en la salida estándar:

```
Palo p;

p= Oros;
cout << "El palo " << p << endl;
```

no obtenemos el nombre que corresponde al valor, sino un número entero. Para poder realizar la lectura y escritura de este tipo usando los identificadores que hemos enumerado, tendremos que crear código adicional para realizar la correspondencia entre el valor entero y el nombre. Por ejemplo, para escribir el palo podemos usar:

```
Palo p;

// ...
switch (p) {
   case Oros:    cout << "Oros"; break;
   case Copas:   cout << "Copas"; break;
   case Espadas: cout << "Espadas"; break;
   case Bastos:  cout << "Bastos"; break;
}
```

2.4.1 Representación del tipo enumerado

La mejor manera de entender el funcionamiento del tipo enumerado y por qué se comporta así es describiendo cómo lo maneja el compilador. Su representación está muy cerca de los tipos integrales. Conocer esta representación nos permite aprovechar esa estrecha relación con los valores enteros propios de los tipos integrales.

La forma en que maneja el compilador un tipo enumerado es muy similar a la solución que podríamos darle si no tuviéramos la posibilidad de definirlos. Por ejemplo, si quiere manejar un objeto que corresponda al palo de una baraja, podemos usar algún tipo integral y varias constantes para cada uno de los valores. Por ejemplo, podríamos definir:

```
typedef int Palo;       // Palo es sinónimo de int
const int Oros    = 0; // También const Palo Oros= 0;
const int Copas   = 1;
const int Espadas = 2;
const int Bastos  = 3;

Palo p; // Realmente contiene un entero
```

Con esta definición, el código anterior que usa el enumerado `Palo` hubiera funcionado exactamente igual. Realmente, cuando el compilador encuentra la definición de un tipo con **enum**,

soluciona el problema realizando una operación similar. Podemos suponer que primero selecciona un tipo integral que subyace bajo el nombre del nuevo tipo, y luego define una constante para cada valor enumerado.

No debe entender que el compilador crea un sinónimo de un tipo *integral* para representar el tipo enumerado. No es un sinónimo, porque el tipo definido con **enum** es un tipo distinto, diferenciado del tipo integral que se usa para representarlo.

El hecho de que se resuelva de esta forma también afecta al lenguaje, que incluye la posibilidad de indicar exactamente los valores enteros que corresponden a cada valor enumerado. Por ejemplo, podemos hacer:

```
enum Nota {BIEN=10, REGULAR=5, MAL=0};
```

donde además de definir el nuevo tipo enumerado, hemos indicado el entero que queremos que corresponda a cada uno de los valores que puede tomar el tipo. Observe que en este ejemplo hemos usado mayúsculas para enfatizar que son constantes.

También es posible especificar el valor sólo para algunos de los identificadores. La forma en que el compilador asigna valores a cada uno de ellos es sumar uno al valor del anterior. Si el primero no tiene valor, el compilador usa cero. Así, el tipo `Palo` ha asignado un valor `0` a `Oros` —el primero— y un valor `1`, `2` y `3` —va sumando uno— a `Copas`, `Espadas` y `Bastos`, respectivamente.

Note que el tipo que subyace para representar el enumerado podría cambiar si especificamos ciertos valores. Por ejemplo, el compilador podría optar por usar un **char** —un solo *byte*— para representar el tipo `Palo`. Sin embargo, si asigno un valor concreto que no cabe en un **char**, tendría que optar por otro tipo. En la práctica se maneja algún tipo integral subyacente —que desconocemos— pero con cierto control, ya que el compilador sabe distinguirlo como nuevo tipo.

La relación con tipos enteros es tan explícita que podemos utilizarlos directamente como constantes enteras. Así, puedo realizar `35*BIEN` que resulta `35*10`. Por este motivo, es posible usar los enumerados como una forma de declarar constantes enteras[7]. Por ejemplo:

```
enum {MAXELEMENTOS=100, MAXLONGITUD=500};
```

declara dos constantes con un valor entero. Observe que el nombre del tipo es opcional. En este ejemplo no lo hemos incluido porque no es nuestra intención usarlo.

2.5 Conversiones

Los tipos de datos básicos pueden tener representaciones incompatibles. Por ejemplo, una secuencia de 32 bits que codifica un entero en complemento a 2 es incompatible con una interpretación como número en coma flotante de simple precisión. Ambas son de 32 bits, pero no tienen prácticamente ninguna relación.

Esta incompatibilidad puede sugerir que es imposible mezclar tipos de datos. Efectivamente, no es posible realizar operaciones con estos dos tipos, pero el sistema puede convertir el tipo entero al valor equivalente[8] de tipo real, y realizar la operación entre operandos de igual tipo.

El lenguaje C++ permite la mezcla de tipos de forma automática. El programador puede ignorar la dificultad de la mezcla de representaciones y dejar que el compilador realice las conversiones automáticas que necesite. Sin embargo, no siempre es posible, y en algunos casos será necesario que indiquemos qué tipo de conversión queremos que se lleve a cabo.

[7] Es posible que si lee código C++, encuentre este tipo de constantes dentro de la definición de clases como una forma de declarar constantes enteras. Sin embargo, el tipo enumerado no se ha creado para este objetivo. De hecho, se recomienda el uso de constantes **static**.

[8] Para ser precisos, deberíamos decir convertir al valor aproximado.



2.5.1 Conversiones implícitas

En un programa aparecen distintos tipos que se pueden "mezclar" de forma que se producen conversiones automáticas de unos tipos a otros. El programador no interviene, sino que son las reglas de lenguaje las que establecen qué conversiones se realizan. Algunos ejemplos son:

- Podemos realizar una asignación en la que el tipo asignado no coincida con el de la expresión a asignar.

```
double a;
....
a= 17; // int a double
```

- Los operandos de una suma tienen distintos tipos:

```
int a;
double b,c;
....
c= a+b;
```

- Una función espera un parámetro de un tipo; y pasamos otro distinto:

```
void Funcion (float f, double d);
....
Funcion(12.0,17); // 2 conversiones
```

- Devolvemos un tipo distinto al declarado:

```
unsigned int Funcion ()
{
   return 17;
}
```

Este tipo de conversiones son automáticas, y se denominan *implícitas*, ya que el programador no realiza ninguna indicación sobre la conversión a realizar. A pesar de ello, que se realicen de forma automática no significa que sean correctas. El programador debe conocer exactamente lo que el compilador está realizando para estar seguro de que el resultado es el deseado.

Existen muchas formas de conversión posibles entre distintos tipos, incluyendo casos en los que se pierde información, e incluso que podríamos considerar con poco sentido. No vamos a revisar todas las reglas en este libro (véase por ejemplo C.6 en Stroustrup[36]), aunque es necesario destacar algunas ideas generales que serán suficientes para la mayoría de nuestros problemas.

Conversiones aritméticas

El primer lugar, podemos considerar las conversiones *aritméticas* habituales; es decir, las que se llevan a cabo cuando se utiliza un operador binario con dos operandos de distinto tipo. Por ejemplo, al operar un entero con un flotante.

En general, el programador puede suponer que los tipos se convierten desde tipos más "restringidos" a otros tipos más "generales" con el fin de que se preserven los valores[9]. Por ejemplo, si operamos un **`char`** con un **`int`**, es de esperar que el tipo **`char`** se convierta a **`int`**, ya que tiene un rango más amplio de posibilidades. Además, seguro que se preserva el valor, pues todos los posibles valores que almacena un **`char`** son representables como **`int`**.

Otro caso corresponde a la mezcla de enteros con reales. Si un operando es entero y el otro es flotante, el primero se convierte al segundo. Intuitivamente podemos supone que pasamos de un tipo a otro más general, ya que los flotantes son capaces de representar no sólo valores enteros, sino también con parte fraccionaria. Sin embargo, esta conversión realmente no tiene por qué preservar los valores, aunque es la más "lógica" y, por tanto, la que se realiza. Note que la conversión de entero a flotante no significa que éste tenga un rango de valores superior. Recuerde que los números reales tienen un rango y una precisión. Por ejemplo, si escribe:

[9] Este tipo de conversiones, donde se preservan los valores, se denominan promociones.

```
int a = 1234567890;
float b = a;
a = b;
cout << a << endl;
```

realiza una doble conversión, de entero a real, y de nuevo a entero. En la conversión de **int** a **float** podría perder información. Por ejemplo, en una máquina que tiene 4 bytes para los dos tipos, podría obtener como resultado *1234567936*. Esta pérdida era de esperar: si los dos tienen 32 bits, los dos tienen la misma capacidad, es decir, pueden almacenar 2^{32} posibilidades como máximo. El tipo **float** tiene capacidad para almacenar valores mucho más grandes que los enteros, valores que están entre dos enteros (con parte fraccionaria), incluso probablemente el valor $+\infty$. Lógicamente, habrá otros valores que estando representados por un **int** no pueden estarlo en un **float**, al menos, de forma exacta.

A pesar de ello, se podría considerar que en la práctica no se pierde información. Recuerde que si trabajamos con números reales debemos ser conscientes de que estamos haciendo cálculos aproximados. Si mezclamos enteros y reales es porque queremos trabajar en aritmética de punto flotante y suponemos que las aproximaciones que realiza el tipo escogido son suficientes. En el ejemplo anterior, si no es admisible la representación obtenida, deberíamos seleccionar un tipo con más capacidad como el tipo **double**. Recordemos que normalmente el valor que escribimos en la salida estándar es el resultado de un redondeo con muchos menos decimales de los que es capaz el tipo, por lo que probablemente veamos un valor que nos parecerá exacto.

Pérdida de información

La pérdida de información en las conversiones no sólo se da cuando la precisión de un flotante aproxima el valor de un entero. También es posible realizar conversiones implícitas donde es clara la posible pérdida de información. Lógicamente, el programador es el responsable de que sean correctas, es decir, el responsable de que no afecten a la validez del algoritmo que se implementa.

Por ejemplo, podemos asignar un **double** a un **int**. En este caso, el programador debe ser consciente de que se pierde la parte fraccionaria y, además, debería tener en cuenta que tal vez el valor resultante no está en el rango válido del entero al que se asigna. Por ejemplo, el siguiente código realiza una asignación sin mucho sentido, ya que el valor `1e30` —que es entero— seguro que no puede almacenarse en un **int**:

```
double a= 1e30;
int b;
b= a;
```

Este tipo de conversiones —donde se pierde información— pueden ser detectadas por el compilador, y avisar para que el programador las revise[10].

Ejemplo 2.5.1 Considere la aplicación que se ha comentado anteriormente, en la que se manejan valores de gris de una imagen (almacenados en objetos **unsigned char**). Suponga que se desea calcular el negativo de una imagen. Desarrolle una función que reciba un valor de gris (*x*), y devuelva su negativo (*255-x*).

Una solución podría ser:

```
unsigned char Negativo (unsigned char x)
{
   return 255-x;
}
```

donde vemos que se mezclan tipo **int** con tipo **unsigned char**. Para realizar la operación `255-x`, se convierte el valor de `x` a entero (no hay ningún problema, pues es más general con

[10] Para evitar el aviso, podemos realizar una conversión explícita —como veremos más adelante— para que el compilador conozca que, efectivamente, es una conversión deseada.



un rango más amplio). Finalmente, para devolver el resultado de la resta —que es un entero— se convierte de nuevo a **`unsigned char`**.

Observe que aunque es un rango más pequeño que el de los enteros, el valor seguro que está en el rango *0-255*. Es decir, la conversión se realiza de un tipo de dato más amplio a uno más pequeño. Es posible que el compilador genere algún mensaje de aviso, pero en este caso el programador sabe que el resultado es correcto.

Fin ejemplo 2.5.1 ■

Conversiones booleanas

Finalmente, otro tipo de conversiones a tener en cuenta son las booleanas. Cualquier valor de tipo puntero, entero o flotante se puede convertir a un valor booleano. Los valores distintos de cero se convierten a **`true`**, y el valor cero a **`false`**. Por ejemplo, una función que devuelve si un número es impar puede escribirse:

```
bool Impar(unsigned int numero)
{
   return numero%2;
}
```

Ejercicio 2.2 Considere el siguiente trozo de código:

```
int n;
unsigned char c;
float f;
double d;
bool b;
//...
b= c= (c+n)/(f+d);
```

Indique las conversiones que se están realizando en la expresión de la última línea.

Más adelante veremos que esta idea de hacer corresponder el valor **`false`** al cero se puede extender a otros tipos aún no presentados. Es el caso del tipo puntero, que almacena una dirección de memoria que podría ser nula: vale cero o distinta de cero.

2.5.2 Conversiones explícitas

El programador puede indicar al compilador que realice una conversión de forma explícita. A este tipo de conversión también se denomina *moldeado* —del inglés *casting*— y aunque en la mayoría de los casos no es necesaria, veremos que en algunos problemas es fundamental.

El lenguaje C ofrece un mecanismo para realizar un moldeado con la siguiente sintaxis:

(TIPO) EXPR

de forma que el resultado de la expresión *EXPR* se convierte al tipo final *TIPO*, si es posible. Un ejemplo es:

```
int a;
float b;
...
a= (int) b;
```

donde el valor de `b` se convierte a entero de forma explícita. El resultado de esa conversión —que es de tipo **`int`**— se asigna al objeto `a`.

Esta sintaxis se mantiene en C++, aunque no es recomendable su uso, ya que incluye cualquier tipo de conversión que pueda realizarse, por peligrosa que sea. Es recomendable tener un control sobre el tipo de conversión que se desea, de forma que el programador sea consciente de lo que está solicitando. En C++, se proponen en su lugar 4 operadores de conversión diferentes:

1. **`static_cast`**. Permite la conversión entre tipos "relacionados". El compilador verifica en tiempo de compilación que es posible la conversión (si no es posible, genera un error).
2. **`reinterpret_cast`**. Permite la conversión entre tipos "sin relación". La falta de relación provoca que, en la mayoría de los casos, los resultados dependan de la implementación, y no sea portable.
3. **`const_cast`**. Permite eliminar el calificador **`const`**. Con ello, es posible modificar algo que aparece como **`const`**. Este molde tiene un comportamiento indeterminado en el caso de que el objeto a modificar estuviera originalmente declarado como **`const`**.
4. **`dynamic_cast`**. El compilador verifica que es posible la conversión en tiempo de ejecución. Se utiliza para realizar conversiones entre clases base y derivadas, ya que no pueden verificarse en tiempo de compilación.

No vamos a estudiar todos estos operadores en este tema. Sin embargo, presentamos el primero de ellos como un operador que podemos usar para algunas conversiones entre tipos aritméticos, evitando la sintaxis de C. Su sintaxis (que como verá, es similar a las de los otros operadores) es:

static_cast<TIPO>(EXPR)

de forma que el valor resultado de *EXPR* se convierte al tipo *TIPO*. Un ejemplo:

```
int a;
float b;
...
a= static_cast<int>(b);
```

Note que las conversiones implícitas que pueden generar avisos en el compilador dejarán de generarlos si se realizan de forma explícita (el programador confirma con ello que es lo que desea).

Además, las conversiones explícitas permiten realizar operaciones que no se pueden realizar de forma implícita. En el caso de que una conversión sea propensa a errores, el compilador no admite la conversión automática. Tiene que ser el programador el que indique explícitamente que la desea. Un ejemplo es la conversión de entero a tipo enumerado. De forma implícita tiene poco sentido, ya que sería fácil equivocarnos escribiendo un entero que asigna a un enumerado un valor no deseado; incluso podría ser desastroso si está fuera del rango de posibles valores pues tendría un valor sin sentido. Si se hace de forma explícita, el programador es más consciente de lo que está haciendo, controlando mejor los casos en los que se realiza esta conversión.

Finalmente, es interesante indicar que también es válida la siguiente sintaxis:

TIPO(EXPR)

como la que presentan las llamadas explícitas a constructores de clases. En principio podemos obviar esta posibilidad, aunque cuando estudie el tema de *clases*, descubrirá que esta sintaxis nos permite usar los tipos simples de igual forma que las clases definidas por el usuario. Un ejemplo concreto de uso de esta sintaxis es el siguiente código:

```
char c;
cin >> c;
cout << "El valor entero es " << int(c) << endl;
```

que convierte el **`char`** *c* a tipo **`int`**. El resultado es que no escribe el carácter correspondiente en la salida estándar, sino el código ASCII que le corresponde ya que a **`cout`** le llega un entero, no un carácter.

