BORRADOR — versión no oficial, compilado el **11-11-2026** (**9d173f5**)

Introducción a la programación en C++

Fundamentos de la programación de ordenadores usando C clásico con algo de C++

Dr. Juan Falgueras Cano
Universidad de Málaga · 2026

Contenidos

Intro La Programación de ordenadores como arte ... 3
0·1 ¿Pero qué es programar? ... 3
0·2 Los LLM o «asistentes» de IA ... 5
0·3 Qué hace que un ordenador sea programable ... 6
0·4 Elegir un lenguaje: ¿por qué C++? ... 7
0·5 El primer ejercicio ... 8
Part 1 Informática y Programación ... 11
1·1 Informática y Programación ... 11
1·2 Estructura Funcional de los Ordenadores ... 12
1·3 Codificación de la Información ... 14
1·4 Algoritmos y Resolución de Problemas ... 17
1·5 Lenguajes de Programación ... 23
1·6 Visión General de un Sistema Informático ... 29
▸ Tarea 0: Editar y compilar nuestro primer fichero fuente en C++ ... 33
Part 2 Introducción a un Lenguaje de Programación ... 34
2·1 Introducción a C++ ... 34
2·2 Tipos de Datos Simples ... 37
2·3 Constantes, Variables y Asignaciones ... 41
2·4 Caracteres y cadenas ... 52
2·5 Enumeración de los valores que puede contener un tipo ... 54
2·6 typedef: Definición de nuevos nombres para tipos ... 55
2·7 Números aleatorios ... 57
2·8 Precedencia de operadores ... 57
2·9 Entrada y Salida Básicas ... 59
▸ Tarea 1: Fundamentos. Programas secuenciales: aritmética y E/S estándar ... 62
2·10 Flujo de control ... 64
2·11 Expresiones booleanas ... 64
▸ Tarea 2: Booleanos ... 68
2·12 Estructuras de selección ... 71
▸ Tarea 3: Estructuras de control. Condicionales ... 80
2·13 Estructuras de iteración ... 83
2·14 Gestión de errores y excepciones ... 89
2·15 Recomendaciones y errores frecuentes ... 91
▸ Tarea 4: Estructuras de control. Bucles simples ... 94
▸ Tarea 5: Estructuras de control. Bucles dobles ... 98
Part 3 Abstracción Procedimental ... 102
3·1 Diseño Descendente ... 102
3·2 Procedimientos y Funciones. Parámetros ... 103
3·3 Recursión ... 119
▸ Tarea 6: Subprogramas ... 128
▸ Tarea 7: Subprogramas++ ... 132
▸ Tarea 8: Pre-estructuras ... 136
Part 4 Tipos de Datos Estructurados ... 138
4·1 Paso de Parámetros con Tipos Estructurados ... 139
4·2 Arrays ... 140
▸ Tarea 9: Arrays ... 149
▸ Tarea 10: Arrays++ ... 156
4·3 Cadenas de caracteres ... 161
▸ Tarea 11: Cadenas de caracteres ... 166
4·4 Estructuras o registros ... 175
▸ Tarea 12: Estructuras ... 180
4·5 Matrices ... 185
▸ Tarea 13: Matrices ... 189
▸ Tarea 14: Problemas de exámenes finales ... 195
Part 5 Manejo de ficheros ... 202
5·1 Declaración de un manejador para manipular un fichero ... 203
5·2 Apertura y cierre del fichero ... 203
5·3 Lectura y escritura de un fichero de texto ... 204
5·4 Los flujos como parámetros de subprogramas ... 206
Part 6 Gráficos, sonido y WIMP con C++ ... 208
6·1 Preparación de las herramientas ... 208
6·2 raylib: dibujo, animación y sonido ... 210
6·3 FLTK: ventanas, botones y eventos ... 213
Part 7 Estilo de programación ... 216
7·1 Comentarios de grandes programadores ... 216
7·2 Estilo y requisitos ... 218

Intro La Programación de ordenadores como arte

Contenido de esta parte Contenido del libro

0·1 ¿Pero qué es programar? 3
0·2 Los LLM o «asistentes» de IA 5
0·3 Qué hace que un ordenador sea programable 6
0·4 Elegir un lenguaje: ¿por qué C++? 7
0·5 El primer ejercicio 8

0·1 ¿Pero qué es programar?

¿Alguna vez te has preguntado cómo funcionan los ordenadores? Este es un curso para principiantes, diseñado para quienes aprenden a programar por primera vez.

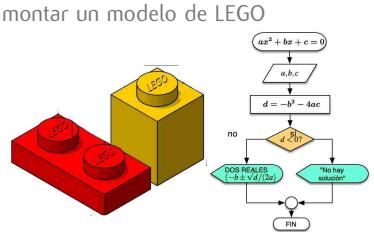
Los ordenadores ejecutan programas, pero ¿qué es programar?

> *Programar es el proceso de instruir a un ordenador para que realice tareas según especificaciones expresadas en un lenguaje concreto*

Programar es fundamentalmente una actividad creativa. Una vez que disponemos del mecanismo adecuado, podemos dirigir al ordenador para que alcance nuestros objetivos mediante múltiples enfoques distintos. Las acciones pueden expresarse de numerosas maneras diferentes y, en consecuencia, aprender a programar es análogo a aprender una forma de arte o adquirir un nuevo idioma.

Cuando vemos a un niño usando piezas de **LEGO** para construir un castillo con esos bloques y todas esas piezas prefabricadas y figuritas, podemos hacernos una idea de lo que realmente es programar. Podemos organizar las piezas de muchas maneras diferentes y eso solo para construir muros. Podemos ser creativos en los detalles, eligiendo distintos tipos de piezas, de diferentes longitudes y formas, para construir algo específico, pero también podemos ser creativos en la organización de las partes más grandes sabiendo que el objetivo final puede alcanzarse con esos módulos o con otros alternativos. La principal diferencia es que la construcción final de LEGO no está pensada para ser entendida por un ordenador, sino por la imaginación del niño.

1: Construir un programa es similar a montar un modelo de LEGO



Cuando los artistas pintan una escena cotidiana, pueden hacerlo de muchas maneras diferentes. Cada una muestra partes del original, la experiencia de quien la captura, que es siempre subjetiva. También añadimos la creatividad y la habilidad del pintor para mostrar no solo lo que hay, sino también lo que la escena puede transmitir a través de la pintura. Cuando hacemos que un ordenador haga algo, aunque no sea tan complejo como la naturaleza, podemos hacer que haga cosas diferentes

de maneras diferentes. Esto se debe a que podemos añadir más capas a las acciones, y podemos elegir entre muchas formas de conectarlas, igual que un niño jugando con LEGO. Así pues, en última instancia, la elección de ellas es un arte que puede reflejar una manera subjetiva de expresar nuestra comprensión del proceso que el ordenador ha de realizar.

James Flynn (Flynn 2014) argumenta que la programación de ordenadores es tanto una forma de arte como una disciplina científica. Apoyándose en Paul Graham y Donald Knuth, traza un paralelismo entre programar y pintar: ambos parten de un espacio en blanco, se refinan hacia una visión creativa y producen una obra para un público humano. Prácticas como el live coding y el concurso IOCCC muestran además cómo el código puede ser expresivo y estético, no meramente funcional. Flynn concluye que programar es accesible para cualquiera que tenga un ordenador y una idea: como un pintor o un compositor, el programador llena una página en blanco y, mediante la iteración, llega a algo elegante y bello. Enseñar a alguien a programar es comparable a enseñarle a pintar obras maestras, aunque con un mayor énfasis en los puntos y coma y una menor atención a las boinas. Podría argumentarse que toda disciplina creativa, incluidas las matemáticas, incorpora cierto grado de arte.

Esto está relacionado con la idea de que «no hay límite a la complejidad que puede alcanzar todo, ya que una cosa siempre lleva a otra». Así, la complejidad de la cadena de acciones puede lograr objetivos inimaginables, tengan sentido o no.

Para D. E. Knuth, programar es algo más que una materia técnica; es un oficio que combina habilidades matemáticas, creatividad para resolver problemas y comunicación orientada a las personas, creando cosas que pueden valorarse por su belleza, claridad y eficiencia. Por eso tituló su ensayo de 1974 en la revista **Communications of the ACM**: «Computer Programming as an Art».

Cuando programas, básicamente estás eligiendo una secuencia definitiva de decisiones y acciones para alcanzar un objetivo. Y, por supuesto, hay decenas de maneras de hacerlo, cada una con su propia mezcla encantadora de ventajas, inconvenientes y «¿por qué diablos elegí este enfoque?».

Sin embargo, los teoremas matemáticos se rigen por reglas diferentes. Cuanto más complejo es el teorema, menos maneras hay de demostrarlo, y por «menos» quiero decir «posiblemente solo una, y que tengas suerte encontrándola». No hay ningún ordenador que valide nuestras demostraciones; el único hardware disponible es nuestro propio cerebro, que es notoriamente imperfecto. Los teoremas más sencillos pueden demostrarse de muchas maneras, pero cuando las cosas se complican, la línea entre «comprensión profunda» y «una suposición fundamentada que desesperadamente esperamos que sea correcta» se vuelve sorprendentemente delgada.

Puedes pedirle a un LLM que escriba tu código por ti, siempre que le proporciones un prompt suficientemente detallado. En ese punto, la «creatividad» se reduce principalmente a lo ingeniosamente que formules tu petición. El modelo rellenará los huecos con entusiasmo, generando código cosido a partir de patrones que ha visto antes. Tu trabajo pasa

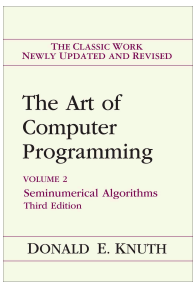
2: Las Meninas (Diego Velázquez), una obra maestra del arte



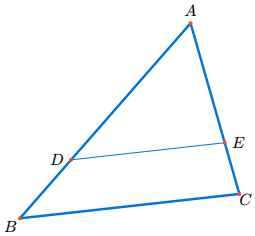
3: La programación de ordenadores es un arte porque aplica el conocimiento acumulado al mundo, porque requiere habilidad e ingenio, y sobre todo porque produce objetos de belleza. Un programador que se ve a sí mismo inconscientemente como un artista disfrutará de lo que hace y lo hará mejor. —D.E. Knuth



4: Este libro (TACP vol 2, año 1969) de D.E. Knuth es y ha sido una referencia para todos los libros posteriores sobre programación.



5: Un teorema matemático, como el teorema de Tales, suele expresarse de una manera y con una notación muy específicas, bastante distintas de las diversas formas en que los lenguajes de programación o el lenguaje natural pueden representarlo



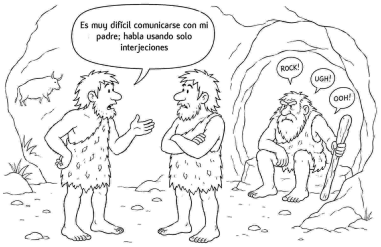
$$\frac{AD}{AB} = \frac{AE}{AC} = \frac{DE}{BC}$$

de programar de verdad a describir vagamente lo que quieres — básicamente el equivalente tecnológico de señalar un catálogo y decir: «*Sí, me llevo ese*», en lugar de coger las herramientas y construirlo tú mismo.

Aunque este enfoque puede **ahorrar tiempo**, no produce necesariamente resultados de mayor calidad. Lo más probable es que elimine alternativas mejores que normalmente requerirían más tiempo de desarrollo.

Al programar en un lenguaje de programación, mantenemos la atención en nuestro objetivo durante un período prolongado, lo que nos brinda la oportunidad de considerar enfoques alternativos. Programar suele ser una actividad que consume tiempo, lo que explica el considerable éxito de los modelos de lenguaje de gran tamaño para ayudar a los programadores en el desarrollo de código; sin embargo, reducir el tiempo no siempre es la estrategia óptima. Para un programador, el tiempo invertido en construir código es el tiempo durante el cual su mente se ajusta y descubre — a menudo de manera inesperada— mejores alternativas, dando como resultado un código que en última instancia es más adecuado para el objetivo específico que el programador tenía en mente.

6: Elegir el lenguaje correcto (sintaxis, gramática, etc.) es fundamental para expresar ideas con precisión



0·2 Los LLM o «asistentes» de IA

Es completamente natural que estos nuevos asistentes nos generen cierta inquietud a todos: a ti, porque puede parecer que todo ya está hecho; a mí, porque quiero que salgas de este curso siendo aún capaz de pensar las cosas por ti mismo. Usaremos este curso para practicar exactamente esa habilidad.

¿Cómo deberíamos usarlos? Con mucha cautela, y siempre de una manera que refuerce, en lugar de reemplazar, tu propio pensamiento. En este curso, nuestra regla de oro será sencilla: durante las tareas evaluadas debes apoyarte en tu propia comprensión, y durante la práctica solo podrás usar asistentes después de haber intentado resolver el problema tú mismo y ser capaz de explicar lo que la herramienta sugirió y por qué.

7: Algunos de los servidores de LLM más populares



Lo que la inteligencia artificial puede hacer con el pensamiento humano es preocupante. Por un lado, puede alcanzar soluciones a problemas extremadamente complejos a una velocidad asombrosa; por otro, corre el riesgo de disolver nuestros propios procesos de pensamiento. Para muchas personas, si la IA resuelve los problemas que antes les importaban, esos problemas dejan de tener sentido, y el compromiso cognitivo que la resolución de problemas proporcionaba sencillamente desaparece.

En este curso, quiero que descubras ese compromiso cognitivo con la programación. Si cada desafío que te planteas se disuelve al instante gracias a un asistente, nunca te das la oportunidad de descubrir que eres capaz de resolverlo por ti mismo. Cuando nunca tienes esa experiencia, pierdes gradualmente el interés en la habilidad: nunca llega a formar parte de tu propio repertorio, no te da ninguna sensación de poder, y por eso simplemente dejas de trabajar con ella.

Leer listas de ejercicios de programación es la forma correcta de aprender a programar en cualquier lenguaje. Los ejercicios cumplen una doble función en dos niveles. En primer lugar, comprender el problema y lo que pide es algo que puedes lograr con relativa rapidez. En segundo lugar, en la práctica, desarrollar las habilidades para resolverlo requiere reflexión,

aunque no necesariamente llegar a una solución completa. Sin embargo, obtienes algo valioso en cada nivel.

0·3 Qué hace que un ordenador sea programable

Los ordenadores pueden realizar una gama tan diversa de tareas porque son programables. Esto significa que los ordenadores no están diseñados para realizar una función única y fija, sino para ejecutar cualquier tarea especificada por sus programas.

Cuando encendemos una luz en casa, accionamos un interruptor que conecta los cables correctos y ocurre la magia. No es muy diferente de lo que hacemos cuando programamos ordenadores. Excepto que los ordenadores son un poco más complicados que un interruptor doméstico. Los ordenadores también tienen cables, muchos cables — tantos que los primeros ordenadores necesitaban edificios enteros para albergarlos.

8: La programación digital en su nivel más fundamental es la conmutación controlada de señales eléctricas entre estados de encendido y apagado



Los ordenadores modernos, e incluso los pequeños relojes inteligentes digitales, son tan complicados que necesitarían todo un edificio de ordenadores del estilo antiguo lleno de cables para hacer lo mismo. E incluso así serían mucho más lentos, porque el tamaño importa. Los electrones tardan en viajar, y cuanto menor es la distancia, menos tiempo se necesita.

Otra diferencia es que con los ordenadores no nos interesa encender y apagar luces. Normalmente lo primero que queremos hacer es mostrar cosas en pantallas, con colores vivos y mucha información brillante. Así que empecemos calculando algo, como una suma, como `2 + 2`. Por tanto, no queremos usar nuestro lenguaje de programación para accionar un interruptor, sino sumar dos números y luego mostrar el resultado de esa suma en la pantalla.

A la inversa, la distinción fundamental de un ordenador respecto a la intrincada red de cables e interruptores comentada anteriormente radica en su capacidad para propagar cambios de estado de forma automática. Cuando se activa o desactiva una conexión, los transistores siguen una lógica cableada para activar o desactivar otras conexiones en respuesta. Usando la analogía de los cables e interruptores, los transistores funcionan como mecanismos que, cuando se activan, conectan y desconectan simultáneamente múltiples componentes adicionales, que a su vez afectan a muchos otros como consecuencia. Este efecto en cascada se origina a partir de una única acción inicial.

9: Programación: del concepto abstracto a la implementación digital





No se trata únicamente de acciones, sino también de estados. Tras ejecutar las acciones, los estados resultantes persisten, se almacenan en la memoria del ordenador y posteriormente influyen en cómo se comportan las acciones posteriores en función de los estados previamente registrados.

Cuando escribimos una instrucción o una oración compuesta por varias de ellas, estamos modificando estados de memoria. Los denominamos variables, estructuras de datos, etc. Muchas operaciones producen efectos en dispositivos externos y periféricos o transmiten acciones a otros ordenadores en red; en consecuencia, una vez que las ejecutamos, perdemos el control sobre el resultado, ya que se encuentra fuera de nuestro sistema.

Las tareas de computación pueden expresarse en distintos niveles de abstracción. El nivel más alto es la imaginación humana, que genera

conceptos y escenarios que articulamos mediante el lenguaje natural, descendiendo eventualmente hasta los lenguajes de programación. Sin embargo, existen múltiples lenguajes de programación para expresar acciones computacionales, cada uno con distintos grados de complejidad. Algunos lenguajes demandan considerablemente más recursos computacionales que otros, dependiendo de su nivel de abstracción. Las abstracciones de nivel superior generalmente requieren más instrucciones de bajo nivel para ejecutarse.

0·4 Elegir un lenguaje: ¿por qué C++?

El lenguaje que nos ocupa, C++, es un lenguaje de programación de nivel relativamente bajo. Cada una de sus instrucciones básicas y predeterminadas produce un número relativamente pequeño de operaciones elementales del ordenador.

El siguiente programa en C++ calcula la suma $2 + 2$ y, lo que es más importante, imprime el resultado en la pantalla:

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

int main() {
  cout << 2 + 2 << endl;
  return 0;
}
```

Sin embargo, un simple cambio como este afecta a la interpretación e imprime no la suma aritmética, sino la cadena de texto literal encerrada entre comillas, es decir: `2 +2`

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

int main() {
  cout << "2 +2" << endl;
  return 0;
}
```

Varios lenguajes de programación ampliamente utilizados tienen una sintaxis similar a este, mientras que otros ofrecen alternativas más simples y menos verbosas que admiten una sintaxis minimalista:

```
puts 2 + 2
```

Esto le dice al **ordenador** «imprime el resultado de la suma 2 + 2 en la pantalla». Tranquilo.

Actualmente existen muchos lenguajes de programación, algunos relativamente modernos, otros clásicos, como C. Entre los más populares en rankings recientes están Python, C++, Java, C#, JavaScript, C, Go, Rust, TypeScript y Swift. **20 Most Popular Programming Languages in 2026**

Así pues, queda claro que, al igual que los idiomas naturales, hablar con los ordenadores y pedirles que hagan cosas es algo «nacionalista». Aquí no hay banderas, pero los lenguajes definen grupos de personas y, por tanto, inevitablemente rivalidad.

La razón de la existencia de tantos lenguajes es que resulta fácil para los programadores definir nuevos lenguajes y construir traductores para

```C++
// C++
#include <iostream>
using namespace std;

int main() {
  cout << 2 + 2 << endl;
  return 0;
}
```

```cs
// C#
using System;

class Program {
  static void Main() {
    Console.WriteLine(2 + 2);
  }
}
```

```java
// Java
public class Main {
  public static void main(String[]
  args) {
    System.out.println(2 + 2);
  }
}
```

```C
// C
#include <stdio.h>

int main() {
  printf("%d" << endl, 2 + 2);
  return 0;
}
```

```Go
// Go
package main

import "fmt"
func main() {
  fmt.Println(2 + 2)
}
```

```Rust
// Rust
fn main() {
  println!("{}", 2 + 2);
}
```

```Python
# Python
print(2 + 2)
```

```JavaScript
// JavaScript
console.log(2 + 2);
```

```TypeScript
// TypeScript
console.log(2 + 2);
```

```swift
// Swift
print(2 + 2)
```

```Ruby
# ruby
p 2 + 2
```

ellos. Por otro lado, está claro que algunos lenguajes son más adecuados que otros para hablar de ciertos temas para los que otros lenguajes son menos expresivos. Si quieres construir matrices, o trabajar con texto, o construir rápidamente aglomeraciones complejas de información, los lenguajes deben seguir sintaxis diferentes.

C y C++ son lenguajes de programación clásicos que siguen siendo relevantes hoy en día porque son eficientes y ofrecen a los programadores un control preciso sobre lo que hace el ordenador.

La simplicidad de C proviene de su estrecha conexión con la forma en que los ordenadores ejecutan realmente las instrucciones. Sus construcciones básicas se corresponden estrechamente con operaciones fundamentales, como almacenar valores en memoria y controlar el flujo de ejecución. Por esta razón, C se utiliza ampliamente para escribir programas que deben ejecutarse muy rápidamente, como partes de sistemas operativos y otros programas fundamentales que sirven de soporte a aplicaciones más complejas.

Así pues, las principales ventajas de elegir C como lenguaje de aprendizaje son su estabilidad y simplicidad. Pero, como es sabido, todo lo bueno tiene su lado malo. Su simplicidad hace más difícil construir cosas complicadas para las que otros lenguajes tienen comandos directos y el nuestro requiere una larga lista de instrucciones delicadas. La estabilidad tiene la desventaja de que su antigua sintaxis es difícil de adaptar a ideas modernas, ya que cualquier cambio en ella supondría incompatibilidad con el enorme conjunto de código ya escrito en C.

0·5 El primer ejercicio

El primer ejercicio consiste necesariamente en editar un programa sencillo, compilarlo (el proceso de traducir nuestro lenguaje, C++, a código legible por la máquina) y, finalmente, ejecutarlo para verificar que se comporta exactamente como se pretende.

Para ello, debes configurar un entorno de programación en tu ordenador. El esfuerzo requerido depende de tu nivel de habilidad y experiencia previa. Algunos de vosotros puede que ya tengáis todo instalado. Editar el archivo fuente suele ser menos exigente que los pasos posteriores de compilación y ejecución. Al abordar un nuevo ejercicio de programación, te encontrarás con tres tareas principales:

1. Crear el archivo fuente. Esta es principalmente una tarea logística en tu ordenador:
   a. Selecciona un directorio apropiado. Suponiendo que guardarás múltiples archivos fuente de C++ en él, el directorio debe estar vacío y tener un nombre descriptivo, como `00.BasicExercises`
   b. Debes inicializar tu entorno de programación. Tu elección de entorno depende de la arquitectura de tu ordenador o sistema operativo: macOS, Linux o Windows. En macOS, recomendaría Terminal + Editor o Xcode. En Windows, **Code::Blocks** es adecuado. En Linux, para quienes ya están familiarizados con él, existen numerosas opciones, aunque Terminal + Editor sigue siendo quizás la más instructiva.

   c. Crea el archivo `01.sum1to100.cpp` y guárdalo (aunque esté vacío) en el directorio, luego escribe el código
2. Compílalo, corrigiendo los errores del compilador a medida que aparezcan. Recuerda guardar tras cada corrección y luego volver a compilar.
3. Ejecuta la aplicación compilada —tu primer programa—, que estará ubicada junto a tu archivo fuente.

Después de dominar estos pasos como secuencia estándar para cada ejercicio, intenta modificar el programa fuente y repite los pasos 2 y 3. Experimentar con modificaciones de esta manera es la forma más efectiva de entender lo que hace el programa.

Procediendo con el ejercicio, copiemos el siguiente programa (no es necesario que lo entiendas en esta fase). Completa los comentarios al principio con el nombre del archivo, tu nombre o alias, la fecha y, si lo deseas, lo que crees que hace el programa:

```cpp
// sum1to100.cpp
// juanfc 2026-09-09
// This program sums all natural
// numbers from 1 up to 100

#include <iostream>
using namespace std;

int main() {
  int sum = 0;
  int i = 1;

  while (i <= 100) {
    sum = sum + i;  // add the current value of i to the sum
    i = i + 1;      // increment loop variable
  }

  cout << "The sum from 1 to 100 is: " << sum << endl;

  return 0;
}
```

El segmento inicial de cada programa contiene invariablemente

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

int main() {
```

cuyo propósito es conectar el entorno en el que estamos desarrollando nuestro programa: un terminal de ordenador con teclado y pantalla para entrada y salida. Sirve para incluir las referencias necesarias a los operadores requeridos para interactuar con estos dispositivos. Lo examinaremos con más detalle más adelante.

1. Modifica el límite `100` a diferentes valores, recompilando y ejecutando el programa tras cada cambio. Puede ser más práctico reducir el límite a `10` o `5` para verificar los resultados.
2. Intenta sumar no los números naturales $(1 + 2 + 3 + ...)$ sino sus cuadrados: $1^2 + 2^2 + 3^2 + ...$
3. Intenta sumar no todos los números naturales sino solo los números naturales impares: $1 + 3 + 5 + ...$

0·5.1 Punto de control

1) ¿Cuál es el significado del símbolo `//` al principio de ciertas líneas y a continuación de algunas instrucciones en el código C++?
2) Las instrucciones iniciales tras la entrada al programa (después de `int main()`) declaran dos variables, `sum` e `i`, y les **asignan** valores (`0` y `1`). ¿Qué ocurriría si omitiéramos esta inicialización, o si asignáramos valores diferentes a estas variables en su lugar?
3) Hay una larga instrucción que concatena múltiples elementos para su salida en pantalla usando `cout << ...`. ¿Por qué algunas partes están encerradas entre `"` (comillas) mientras que otras, como `sum` o `endl`, no lo están? ¿Qué las distingue?
4) La instrucción `return 0;` indica al sistema operativo que el programa se ejecutó correctamente. ¿Puedes imaginar cómo podríamos señalar al sistema operativo que nuestro programa encontró problemas al salir?

Referencias

FLYNN, James, 2014. Exploring computer coding as an art form. *The Varsity*. Online. 17 marzo 2014. Available from: thevarsity.ca [Accessed 11 junio 2026].

Part 1 Informática y Programación

Contenido de esta parte Contenido del libro

1·1 Informática y Programación 11
1·2 Estructura Funcional de los Ordenadores 12
1·3 Codificación de la Información 14
1·4 Algoritmos y Resolución de Problemas 17
1·5 Lenguajes de Programación 23
1·6 Visión General de un Sistema Informático 29
▸ Tarea 0: Editar y compilar nuestro primer fichero fuente en C++ . 33

1·1 Informática y Programación

La Informática no es el estudio de los ordenadores, del mismo modo que la astronomía no es el estudio de los telescopios:

> es la ciencia de **qué**
> puede computarse y **cómo**

10: Programar implica usar ordenadores para modelar problemas y observar los resultados, del mismo modo que los astrónomos usan telescopios para comprender el universo



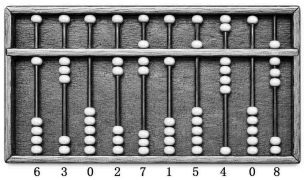
Es a la vez una *ciencia*, que estudia qué puede computarse, y una disciplina de *ingeniería*, que estudia cómo construir programas que lo computen. Por otro lado,

> **programar** es el oficio que transforma
> ideas en artefactos funcionales,

y será nuestra herramienta principal a lo largo del curso.

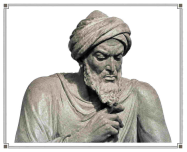
El deseo de calcular es mucho más antiguo que las máquinas. Durante miles de años, mercaderes y astrónomos realizaron sus cálculos aritméticos con el **ábaco**, deslizando cuentas a lo largo de varillas: una memoria de madera donde los dedos ejecutaban el procedimiento. La herramienta no pensaba; la persona que la operaba seguía reglas fijas, paso a paso. Esa idea — *un procedimiento mecánico que cualquiera puede repetir y que siempre da la respuesta correcta* — es la semilla de todo lo que se estudia en este curso.

11: El ábaco: un «ordenador» de 4000 años cuyo procesador era la persona que lo usaba



En el siglo IX, el matemático persa Muhammad ibn Musa **al-Juarizmí** escribió tales procedimientos para la aritmética y la resolución de ecuaciones. Cuando sus obras fueron traducidas al latín, su nombre se convirtió en *algorismus* — el origen de nuestra palabra **algoritmo** — y el título de uno de sus libros nos legó la palabra *álgebra*. Así, el concepto central de la computación recibió su nombre por una persona doce siglos antes de que existiera el primer ordenador.

12: Al-Juarizmí (c. 780–850), cuyo nombre latinizado nos dio la palabra: *algoritmo*

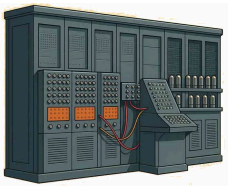


Las máquinas fueron asumiendo gradualmente la parte mecánica. Pascal (1642) y Leibniz construyeron calculadoras de engranajes; en la década de 1830, Charles **Babbage** diseñó — aunque nunca terminó — la *Máquina Analítica*, la primera máquina concebida para ser programable, y **Ada Lovelace** escribió lo que generalmente se considera el primer programa

para ella. También realizó una observación notablemente moderna: tal máquina podría manipular no solo números, sino *cualquier* símbolo, incluida la música.

Los pasos decisivos llegaron en el siglo XX. En 1936, antes de que existiera ningún ordenador electrónico, Alan **Turing** respondió sobre el papel a la pregunta de *qué puede computarse en absoluto* — el fundamento teórico de la Informática. Una década después, máquinas como el ENIAC (1945) ocupaban salas enteras y se programaban recableando conexiones. El transistor, y después el microprocesador (1971), redujeron aquellas salas a un chip; los ordenadores personales de los años ochenta pusieron uno en cada escritorio, y hoy el teléfono en tu bolsillo es millones de veces más potente que el ENIAC. Lo que nunca ha cambiado es lo que se necesita para que estas máquinas sean útiles: alguien capaz de expresar un procedimiento con precisión — un **programador**.

13: ENIAC (1945): 30 toneladas, 18.000 tubos de vacío — programado mediante el recableado de conexiones



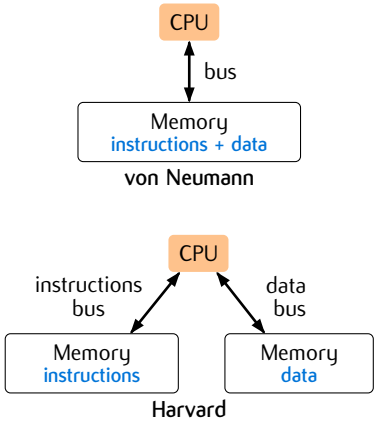
14: John von Neumann, 1903-1957



1·2 Estructura Funcional de los Ordenadores

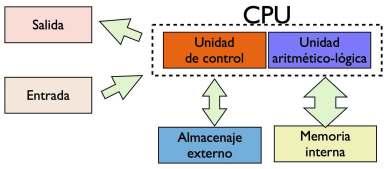
Casi todos los ordenadores, desde el controlador de una lavadora hasta un superordenador, siguen el mismo esquema arquitectónico propuesto por von Neumann: memoria, procesador y unidades de entrada/salida conectadas por buses. El diseño de von Neumann tiene un rasgo definitorio: instrucciones y datos comparten la misma memoria y el mismo bus. Esto es simple y flexible — un programa puede tratar sus propias instrucciones como datos, lo que hace posibles los compiladores, los cargadores y el firmware de actualización automática — pero también implica que el procesador no puede buscar una instrucción y leer un dato al mismo tiempo. Ambos hacen cola en el mismo carril, uno tras otro. Esta limitación se denomina frecuentemente el *cuello de botella de von Neumann*.

15: El cuello de botella de von Neumann



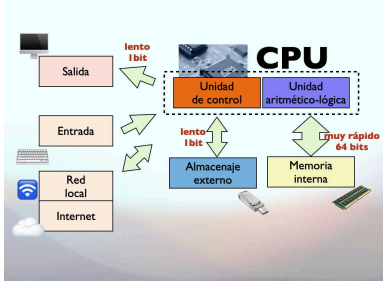
La alternativa clásica es la **arquitectura Harvard**, que mantiene instrucciones y datos en memorias separadas, cada una con su propio bus. No es solo una curiosidad histórica — es la opción por defecto en la mayoría de los microcontroladores y procesadores de señal digital, incluidos los microcontroladores de 8 bits presentes en muchas placas Arduino, precisamente porque los sistemas embebidos valoran la ejecución predecible y rápida por encima de la flexibilidad de un espacio de memoria compartido.

16: La organización de von Neumann: CPU, memoria y dispositivos de entrada/salida



La mayoría de las CPU de propósito general modernas — las que encontramos en portátiles y teléfonos móviles — emplean en realidad una solución de compromiso conocida como **arquitectura Harvard modificada**. La memoria principal es unificada, como en el modelo de von Neumann, pero el núcleo del procesador está flanqueado por dos pequeñas **cachés** separadas: una para instrucciones y otra para datos. El procesador logra un acceso paralelo al estilo Harvard para la gran mayoría de las operaciones de memoria, que se sirven desde estas cachés, manteniendo al mismo tiempo la simplicidad de un único espacio de direcciones para el resto de operaciones.

17: Tipos de conexiones y velocidad relativa de cada una de ellas



Una consideración contemporánea importante es el grado de paralelismo en el ciclo de búsqueda-decodificación-ejecución que examinaremos (Sección 1·2.1). Una máquina von Neumann clásica ejecuta instrucciones de forma secuencial en un único procesador. El hardware moderno, en

cambio, amplifica este ciclo fundamental: los procesadores multinúcleo ejecutan múltiples flujos de instrucciones independientes de forma concurrente, las unidades SIMD (instrucción única, múltiples datos; puedes leer más sobre esto aquí) aplican una sola instrucción a múltiples valores de datos simultáneamente, y las GPU (unidades de procesamiento gráfico) llevan esto al extremo empleando miles de núcleos simples optimizados precisamente para este tipo de computación paralela de datos, como el renderizado de píxeles o el entrenamiento de redes neuronales. Ninguna de estas arquitecturas abandona el principio de búsqueda-decodificación--ejecución — simplemente ejecutan muchas instancias de él en paralelo.

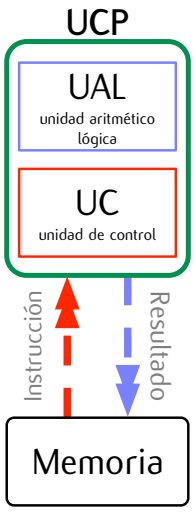
En informática, el paralelismo y la concurrencia son dos conceptos distintos: un **programa paralelo usa múltiples núcleos de CPU**, siendo cada núcleo responsable de una tarea de forma independiente. La **concurrencia**, en cambio, permite a un programa gestionar múltiples tareas incluso en un único núcleo de CPU; el núcleo alterna entre tareas (es decir, **hilos**) sin necesidad de completar cada una antes de pasar a la siguiente. Un programa puede exhibir paralelismo, concurrencia, ambos o ninguno. Ver Wikipedia

1·2.1 Funcionamiento Interno de los Ordenadores

Cuando un ordenador ejecuta un programa, decimos que el ordenador está ejecutando ese programa. La unidad central de procesamiento (CPU), como hemos visto, es el componente responsable de ejecutar los programas. La CPU es el componente más crítico de un ordenador porque sin ella el sistema no puede ejecutar software.

En los primeros ordenadores, las CPU eran dispositivos de gran tamaño construidos con componentes eléctricos y mecánicos como tubos de vacío e interruptores. Las CPU modernas, conocidas como microprocesadores, son pequeños circuitos integrados lo suficientemente pequeños como para caber en la palma de la mano. Además de ser considerablemente más pequeños que las CPU electromecánicas de los primeros ordenadores, los microprocesadores actuales son también significativamente más potentes.

18: Partes principales, muy simplificadas, de un ordenador von Neumann. Las CPU actuales son multinúcleo: muchas de ellas trabajando al mismo tiempo en paralelo



La función de la CPU es obtener instrucciones, decodificarlas y ejecutarlas para producir un resultado. Internamente, la unidad central de procesamiento comprende dos componentes principales: la unidad de control (UC) y la unidad aritmético-lógica (UAL). La unidad de control orquesta todas las operaciones del ordenador determinando la ubicación de la siguiente instrucción y regulando los demás componentes principales del ordenador mediante señales de control. La unidad aritmético-lógica realiza operaciones matemáticas y lógicas, como su nombre indica. La organización de la CPU se ilustra en Figura 18. El ciclo fundamental del procesador consiste en obtener, decodificar y ejecutar instrucciones.

Obtención La unidad de control de la CPU recupera la siguiente instrucción en la secuencia del programa desde la memoria principal. La ubicación de la instrucción viene determinada por el contador de programa (PC), un puntero crítico dentro de la CPU. Cuando los programas se inician, el PC se posiciona en la primera instrucción de `int main() { @` ←. Numerosas instrucciones del procesador permiten que el PC salte de una ubicación a otra. Especialmente

importantes, como veremos, son los condicionales y los bucles, que hacen que el PC salte a otra parte del código en memoria.

Decodificación La instrucción, codificada como un valor numérico, se decodifica: la unidad de control la interpreta y genera las señales de control necesarias para llevarla a cabo.

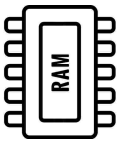
Ejecución Las señales de control se dirigen al componente hardware apropiado (como la UAL, una unidad de disco u otro dispositivo), que a continuación realiza la operación especificada. La UAL en sí solo puede ejecutar operaciones aritméticas y lógicas básicas (`+`, `-`, `*`, `and`, `or`, ...); si la CPU dispone de una unidad de punto flotante (como ocurre en las modernas), la aritmética con números reales se realiza directamente en hardware, de lo contrario se delega a emulación por software.

La memoria del ordenador puede clasificarse según velocidad y capacidad. La memoria más cercana a la CPU es más rápida pero tiene menor capacidad. Las categorías principales son:

Memoria de registros un pequeño número de variables almacenadas directamente dentro de la CPU, accesibles a la velocidad de reloj de la CPU (`A0`, `D7`).

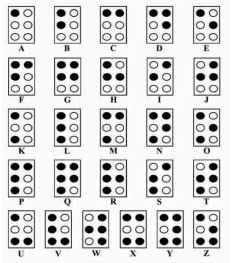
RAM memoria de acceso aleatorio, que proporciona acceso muy rápido, aunque más lento que los registros, requiriendo típicamente varios ciclos de CPU para leer o escribir. Se accede a ella mediante direcciones de memoria — una enumeración secuencial de cada byte de la RAM del ordenador (`0xAFFFFAFFFAFA` es un ejemplo típico de cómo referenciar una posición en memoria; usamos notación hexadecimal).

19: RAM, ROM, EPROM y tecnologías similares son diferentes formas de memoria interna del ordenador



Memoria externa conectada mediante interfaces internas o externas (p. ej., SATA, USB), que ofrece las mayores capacidades al menor coste. El contenido se organiza en bloques denominados ficheros. No nos ocupamos de la ubicación o distribución de esos ficheros en los dispositivos; esa tarea corresponde a los propios controladores de los dispositivos. Solo necesitamos conocer el nombre de los ficheros (`01.myName.cpp`), los directorios en los que se encuentran (`/Users/myName/00.intro/01.myName.cpp`), y a veces sus atributos — si podemos o no leerlos, etc.

20: Codificación Braille de las letras



Repositorios en red e Internet varios órdenes de magnitud más lentos que las opciones anteriores. Se accede a ellos mediante URLs (localizadores uniformes de recursos), que especifican tanto la ubicación del recurso como el protocolo necesario para la comunicación (https://www.uma.es). Al final de cada URL hay un comando para un servidor o un fichero directamente (habitualmente un `file.html`) para descargar e interpretar con nuestro navegador web.

1·3 Codificación de la Información

Los ordenadores solo manipulan señales de dos estados•, por lo que toda información — números, texto, imágenes, sonido — debe codificarse primero en binario. Esta sección examina cómo funciona esa codificación y cuáles son sus límites.

• La razón por la que los ordenadores son binarios y no hexadecimales, por ejemplo, es que un transistor puede distinguir de forma **barata** y **fiable** solo **dos estados de tensión** (alto/bajo). Para almacenar un dígito hexadecimal de forma nativa se necesitaría un componente que mantuviera de forma fiable 16 niveles de tensión distintos — mucho más difícil de construir y bastante más propenso a errores a medida que las tensiones varían con el calor y el ruido.

La alternativa es manejar la información de forma **analógica**: como una magnitud física en correspondencia continua y directa con lo que representa. Eso es lo que hace un termómetro de mercurio o un velocímetro clásico de automóvil — y lo que hacían los discos de vinilo con la música: el surco es una copia física de la propia onda sonora. En principio nada supera esa fidelidad, ya que la digitalización reduce la realidad a un conjunto finito de números. En la práctica, sin embargo, la victoria de lo **digital** ha sido total, y por buenas razones: los pulsos binarios pueden almacenarse, copiarse y transmitirse *exactamente*, millones de veces, sin la degradación que sufre cada copia analógica — los errores incluso pueden detectarse y corregirse automáticamente. El salto del vinilo al disco compacto (1982) hizo audible la diferencia para todos.

21: Discos de vinilo: el surco es una copia física de la propia onda sonora



La información digital tiene una unidad mínima natural: el **bit** (*dígito binario*), un único `0` ó `1`. Un solo bit dice muy poco, por lo que la memoria maneja bits en grupos — ocho de ellos forman un **byte** — y el significado de un grupo depende enteramente del código con el que acordemos leerlo: el mismo patrón de bits puede ser un número, una letra o el color de un píxel.

22: En memoria, cada grupo de bits representa un tipo específico de información



1·3.1 Qué puede almacenar la memoria

El elemento de memoria más simple que puede almacenar un ordenador es `0`/`1`. La única información que puede representar es el valor de verdad de algo: `false` o `true`. Estos se denominan valores booleanos, y representan la forma más simple de información — un único `true` o `false` — pero son fundamentales para la programación. La siguiente forma de información más importante son los números: los enteros y los números reales están presentes en todos los lenguajes de programación, y el hardware de cualquier ordenador soporta directamente la suma, multiplicación y división de enteros.

Para codificar enteros, usamos el antiguo sistema posicional; la única elección es la base — `10` en la vida cotidiana, 2 dentro del ordenador. El valor de un dígito depende de dónde se sitúa: el 3 en 35 no es el 3 en 53. Así, $214 = 2 \cdot 10^2 + 1 \cdot 10^1 + 4 \cdot 10^0$, y exactamente la misma idea en base 2 da, por ejemplo, $1101_2 = 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 13$. Las bases 8 y 16 también son habituales en informática, simplemente porque cada dígito octal o hexadecimal agrupa exactamente 3 ó 4 bits.

Enteros sin signo. Con n bits podemos formar 2^n patrones distintos, por lo que la codificación natural cubre el rango de 0 a $2^n - 1$: un byte almacena de `0` a 255, y 32 bits alcanzan algo más de cuatro mil millones. Nótese algo sin equivalente en matemáticas: el número de bits es *fijo de antemano*. Sumar 1 al valor más grande hace que el resultado dé la vuelta — un **desbordamiento**, fuente clásica de errores en los programas.

Enteros negativos: complemento a dos. En lugar de almacenar un signo separado, los ordenadores dan al bit más a la izquierda un peso *negativo*: en un byte cuenta como -128 en vez de $+128$. Así `10000001` es $-128 + 1 = -127$, y el rango de un byte pasa a ser de -128 a $+127$ (un negativo más que positivos). La elegancia de este truco, llamado **complemento a dos**, es que el mismo circuito de suma funciona igual para números positivos y negativos — sin casos especiales.

$$\mathbb{R} \supset \mathbb{A}_{\mathbb{R}} \supset \mathbb{Q} \supset \mathbb{Z} \supset \mathbb{N}$$

$\mathbb{R}$	Real	$\pi,\ e,\ -2\pi$
$\mathbb{A}_{\mathbb{R}}$	Real Algebraico	$\sqrt{2},\ -\sqrt{3},\ \frac{1+\sqrt{5}}{2}$
$\mathbb{Q}$	Racional	$\frac{1}{2},\ -\frac{2}{3}$
$\mathbb{Z}$	Entero	$-1,\ -2,\ -3$
$\mathbb{N}$	Natural	$0,\ 1,\ 3$

Números reales. Para los reales, los ordenadores usan **coma flotante**: notación científica binaria, que almacena un signo, algunos dígitos significativos (la *mantisa*) y un exponente, en un formato estándar (IEEE 754) que en C++ conocerás como `float` y `double`. El exponente proporciona un rango enorme, pero la mantisa es finita, por lo que casi ningún número real se almacena de forma exacta — ni siquiera un aparentemente inocente 0.1, que en binario es periódico, como $1/3$ en decimal. Todo número real es, en general, una *aproximación redondeada*; tenlo en cuenta, porque comparar dos reales por igualdad exacta es una de las trampas clásicas para principiantes a la que volveremos.

1·3.2 Códigos de E/S. ASCII

Para intercambiar texto con personas, los ordenadores necesitan tablas acordadas que asocien caracteres a patrones de bits. Examinamos las más comunes, desde ASCII (Código Estándar Americano para el Intercambio de Información) hasta Unicode.

HI/LO	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
\n 0	\0	SOH	STX	ETX	EOT	ENQ	ACK	\a	\b	\t	\n	\v	\f	\r	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0x30 3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x41 4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x61 6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

La tabla ASCII comprende cuatro bloques de códigos:

Caracteres de control (`0x00`–`0x1F`, 32 posiciones) caracteres de control residuos de protocolos de comunicación, donde bytes individuales señalizaban funciones como el inicio de un bloque de datos. Ejemplos notables son el salto de línea `\n` (posición `0x0A`), el tabulador (`\t`) y el carácter nulo (`\0`, `0x00`).

Símbolos y dígitos (`0x20`–`0x3F`, 32 posiciones) símbolos estándar incluyendo el signo de exclamación `!`, las comillas, los dígitos (`0` a `9`) y caracteres relacionados.

Alfabeto en mayúsculas (`0x40`–`0x5F`, 32 posiciones) letras mayúsculas (`A` a `Z`) y símbolos adicionales.

Alfabeto en minúsculas (`0x60`–`0x7F`, 32 posiciones) letras minúsculas (`a` a `z`) y símbolos adicionales.

Cuando escribimos cualquier letra, símbolo o dígito en un ordenador, lo que el sistema guarda en memoria, almacena en disco y transmite por Internet es un byte que corresponde a ese carácter según la tabla ASCII.

Sin embargo, habitualmente usamos alfabetos que van mucho más allá del alfabeto latino básico de 26 letras. Usamos caracteres como la ñ española y emojis como 👍, que se codifican en extensiones del estándar ASCII. Una serie de extensiones a este estándar comenzó en 1992 con UTF-8, y hoy podemos codificar miles de símbolos usando múltiples bytes. La tabla ASCII forma un subconjunto de este esquema de codificación más amplio, capaz de representar una gran cantidad de símbolos.

A lo largo de este curso, asumimos que se utiliza la codificación ASCII, donde cada carácter se representa mediante un único byte.

1·3.3 Punto de Control

1) ¿Qué ha hecho que los dispositivos informáticos sean tan útiles y estén tan ampliamente adoptados en diversas profesiones y por un público tan amplio?
2) ¿Cuál es el componente más importante de un ordenador, y qué otros componentes son absolutamente necesarios para su funcionamiento?
3) ¿Cuál es la diferencia clave entre las arquitecturas von Neumann y Harvard?
4) ¿Por qué un microcontrolador podría preferir una arquitectura Harvard (o Harvard modificada) frente a una von Neumann?
5) ¿Qué dos unidades componen internamente la CPU?
6) ¿Cuál es la diferencia entre paralelismo y concurrencia?
7) ¿Puede un programa ser concurrente y paralelo al mismo tiempo?
8) ¿Puede un programa ser paralelo y no ser concurrente?
9) ¿Qué es una dirección de memoria y para qué sirve?
10) ¿Por qué los ordenadores necesitan tanto memoria principal como almacenamiento secundario?
11) ¿En qué categoría de software se incluyen los procesadores de texto, las hojas de cálculo, los programas de correo electrónico, los navegadores web y los juegos?

1·4 Algoritmos y Resolución de Problemas

Antes de escribir una sola línea de código, debemos tener una comprensión precisa del problema que estamos resolviendo y de cómo resolverlo. El algoritmo — un procedimiento finito e inequívoco — sirve de puente entre el problema y el programa. Un algoritmo se convierte en programa una vez que se expresa en un lenguaje de programación. Antes de esta etapa, un algoritmo es más abstracto; puede expresarse de múltiples formas, y solo cuando se formaliza en un lenguaje de programación puede denominarse propiamente programa. Tras muchos años dedicados a enseñar cómo resolver problemas matemáticos, Pólya sintetizó esto en un conjunto de pasos genéricos para resolver cualquier problema:

1. Comprender el Problema

2. Elaborar un Plan
3. Ejecutar el Plan
4. Revisar

Comprender el Problema Antes de escribir ningún código, debemos ser capaces de enunciar claramente qué se da (los datos) y qué se requiere (el resultado), junto con las condiciones que deben satisfacer los datos. Muchos programas fallidos no se deben a una lógica errónea, sino a un problema mal definido: reformularlo con tus propias palabras, identificar entradas y salidas, y trabajar un ejemplo pequeño a mano son las formas más seguras de descubrir ambigüedades antes de que se conviertan en errores.

Elaborar un Plan Una vez comprendido el problema, buscamos una estrategia que conecte los datos con el resultado. Esto rara vez se inventa de la nada: habitualmente implica reconocer el problema como similar a uno ya resuelto, descomponerlo en subproblemas más pequeños, o elegir entre técnicas establecidas — iteración, recursión, divide y vencerás — que se ajusten a su estructura. Un buen plan puede expresarse como pseudocódigo o diagrama (Diagrama de Flujo de Control Figura 25), con independencia de cualquier lenguaje de programación concreto.

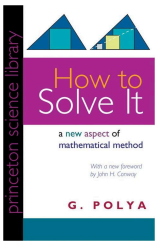
Ejecutar el Plan Solo ahora traducimos el plan a código real, implementándolo paso a paso y comprobando, en cada etapa, que el paso recién dado es correcto antes de continuar con el siguiente. Aquí la paciencia importa más que la velocidad: un plan ejecutado descuidadamente introduce errores que ninguna cantidad de depuración posterior llega a explicar del todo.

Revisar Una vez que la solución funciona, resiste la tentación de considerarla terminada. Examina si el resultado es plausible, pruébala con casos que no usaste durante su construcción (especialmente casos límite), y pregúntate si el mismo razonamiento podría reutilizarse o si la solución podría simplificarse. Este último paso es el que convierte un ejercicio resuelto en experiencia transferible — precisamente lo que separa a un programador que aprende de uno que simplemente termina tareas.

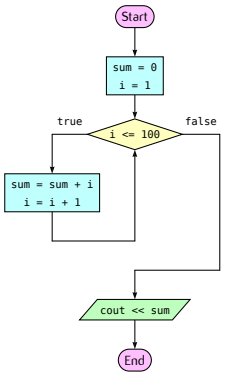
23: George Pólya, 1887-1985



24: Los pasos de Pólya para resolver cualquier problema



25: Diagrama de Flujo de Control del programa en Sección 2·1.2



26: Primer libro sobre algoritmos: «Kitāb al-Jamʿwa-l-tafrīq bi-ḥisāb al-Hind» (El libro compendioso sobre el cálculo por completación y equilibrio), Bagdad, 833 d.C.



27: ¿Cuál es el algoritmo para hacer limonada?



Algoritmo: **Limonada**
1. En una jarra de 2 litros, combina el zumo de 4 limones, agua y azúcar
2. Remueve hasta que el azúcar se disuelva
3. Enfría en el frigorífico.

1·4.1 ¿Qué queremos hacer con los algoritmos?

Como una receta de cocina, un algoritmo comprende una secuencia ordenada de pasos precisos que transforman entradas (datos) en una salida deseada. Definimos las propiedades que debe satisfacer cualquier algoritmo válido. Calcular la altura de una montaña a partir de la longitud y el ángulo de su sombra era uno de los muchos problemas recopilados por Al-Juarizmí.

Todo lo que pueda expresarse mediante una serie de pasos inequívocos y ejecutarse por algún dispositivo informático puede programarse. Sin embargo, veremos que algunos problemas aparentemente razonables — problemas para los que podría parecer que existe un algoritmo — no

tienen en realidad ninguna solución algorítmica factible y lógicamente correcta.

1·4.2 ¿Qué puede hacerse? Computabilidad

Sorprendentemente, algunos problemas bien definidos no admiten ningún algoritmo, mientras que otros tienen soluciones prohibitivamente lentas para ser de utilidad práctica. La **computabilidad** y la **complejidad** delimitan los límites de lo que la programación puede lograr: la primera en cuanto a *viabilidad*, la segunda en cuanto a *eficiencia*.

Algunos problemas, aunque bien definidos, son inherentemente irresolubles. El problema de la parada se clasifica como **indecidible**.

Los problemas que buscan soluciones que implican autoreferencia — como «esta oración es falsa» — están estrechamente relacionados. De forma similar, la pregunta de si el conjunto de todos los conjuntos debe contenerse a sí mismo ejemplifica este asunto. El problema de la parada concierne específicamente a la posibilidad de crear un programa capaz de determinar si cualquier programa arbitrario terminará cuando se le dé una entrada particular. Dado que es de naturaleza autoreferencial, tal programa no puede construirse.

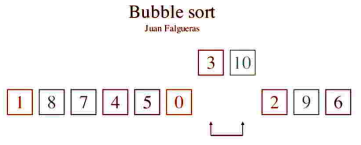
1·4.3 ¿Cuán difícil puede ser? Complejidad

A la inversa, aunque muchos problemas tienen soluciones conocidas, los distintos algoritmos que resuelven el mismo problema no consumen necesariamente los mismos recursos en términos de tiempo de ejecución o espacio de memoria. Estos dos recursos están interrelacionados: algunas soluciones pueden optimizarse en velocidad usando memoria adicional, y viceversa.

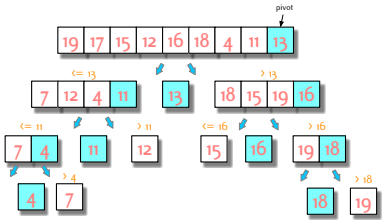
Por ejemplo, si almacenas resultados intermedios calculados durante la ejecución del programa, necesitarás más espacio de memoria; sin embargo, evitarás recalcular los mismos valores repetidamente, reduciendo así el tiempo de ejecución a costa de un mayor consumo de memoria.

No obstante, el rendimiento de distintos algoritmos para resolver problemas idénticos puede variar drásticamente. En consecuencia, el estudio de la complejidad computacional — centrado habitualmente en la eficiencia temporal — constituye un área crucial dentro de la teoría de la computación.

28: La ordenación de burbuja es el algoritmo de ordenación directa más ampliamente utilizado



29: Quicksort, en cambio, es uno de los métodos de ordenación de propósito general más eficientes



Al ordenar una secuencia de valores, se mejora el rendimiento de búsqueda sobre esos valores. La ordenación es una operación fundamental que los ordenadores realizan continuamente, y su implementación eficiente es una preocupación crítica estudiada durante mucho tiempo. Un enfoque es la ordenación por selección: identificar el valor más pequeño y colocarlo en la primera posición, luego avanzar una posición y repetir con los datos restantes. Sin embargo, aunque conceptualmente sencillo, la ordenación por selección funciona mal en la práctica, especialmente con conjuntos de datos grandes. Una alternativa significativamente más rápida es el algoritmo *Quicksort* (atribuido a Hoare), que divide una lista grande en sublistas donde todos los elementos de cada sublista son menores que los de la siguiente, y luego ordena recursivamente estas listas más pequeñas intercambiando elementos desordenados.

Para comparar algoritmos de forma justa — independientemente de la máquina, el compilador o el ánimo del programador del día — no contamos segundos sino **operaciones**, expresadas como función del tamaño de entrada n. Lo que importa no es el recuento exacto sino cómo *crece* a medida que n crece: esto se captura con la notación **asintótica**, o **Big-O**:

$$O(\cdot)$$

que conserva solo el término dominante y descarta factores constantes y detalles de orden inferior. Un algoritmo que realiza $3n^2 + 7n + 12$ operaciones es simplemente $O(n^2)$: para n grande, doblar la entrada aproximadamente cuadruplica el trabajo, y el $+7n + 12$ se vuelve irrelevante.

Desde una perspectiva de complejidad, los problemas pueden clasificarse como:

Tratables tienen una solución eficiente, típicamente alcanzable en tiempo polinomial, $O(n^k)$.

Intratables carecen de una solución eficiente, como el Problema del Viajante, en el que se busca el camino más corto que visite cada ciudad de un conjunto dado exactamente una vez. Este problema está relacionado con numerosos otros, incluyendo el diseño de circuitos impresos, y pertenece a la clase formalmente definida **NP-hard**, a la que aludimos durante nuestra discusión sobre la decidibilidad.

Para los problemas intratables, el enfoque típico de fuerza bruta consiste en evaluar todas las soluciones posibles — cuyo número crece factorialmente ($n!$) o exponencialmente (2^n) con n — y seleccionar la óptima. No hay atajo hacia la respuesta; en cambio, hay que explorar cada camino posible, como en un laberinto sin salida señalizada, comparando exhaustivamente los candidatos para identificar el mejor.

Concretamente: comprobar cada ruta del Problema del Viajante con solo 20 ciudades significa examinar aproximadamente $20! \approx 2.4 \times 10^{18}$ posibilidades. Incluso a mil millones (10^9) comprobaciones por segundo, eso son unos 77 años de computación — para solo 20 ciudades. Ver Figura 30

30: Crecimiento exponencial de los factoriales $n!$ cuando no existe una solución eficiente y es sencillo verificar cada solución candidata. El Problema del Viajante ejemplifica este escenario

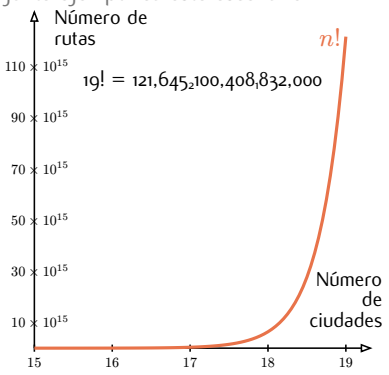
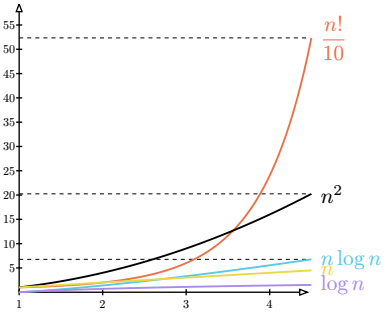


Figura 31 alinea las clases de complejidad habituales de más barata a más cara: $O(\log n)$ apenas crece — piensa en dividir el espacio de búsqueda a la mitad en cada paso, como hace la búsqueda binaria; $O(n)$ es lineal, por lo que duplicar la entrada duplica el trabajo; $O(n \log n)$ es lo que logran los mejores algoritmos de ordenación de propósito general; $O(n^2)$ es típico de comparar cada par de elementos; y $O(n!)$ (con $O(2^n)$ muy cerca) explota tan rápidamente que hace que incluso entradas de tamaño moderado sean computacionalmente inabordables. Nótese cuán cercanas permanecen las primeras cuatro curvas incluso a medida que n crece, mientras que $n!$ se dispara casi de inmediato: ese abismo — no la diferencia entre, digamos, 400 y 4.000 operaciones — es lo que separa un programa que termina instantáneamente de uno que no terminará antes de que el sol se extinga.

31: Crecimiento de $\log n$, n, $n \log n$ (lineal-logarítmico), n^2 y $n!$ a medida que aumenta el tamaño de entrada n



Entre los problemas tratables, el tipo de algoritmo determina su eficiencia. Como se señaló, los algoritmos de ordenación han sido ampliamente estudiados: en el caso general, los simples (ordenación de burbuja, por selección, por inserción) se ejecutan en tiempo cuadrático, $O(n^2)$, mientras que los mejores de propósito general (Quicksort, mergesort, heapsort) se ejecutan en tiempo lineal-logarítmico, $O(n \log n)$ — una diferencia que, según Figura 31, se vuelve dramática bastante antes de que n alcance los millones de registros que podría contener una base de datos real.

1·4.4 ¿Cómo hacerlo? Corrección

Un programa que parece funcionar correctamente no es equivalente a uno que lo hace de forma demostrable. Introducimos el concepto de razonar sobre la **Corrección** en lugar de confiar únicamente en las pruebas y la conjetura. Un campo sustancial dentro de la informática está dedicado a la corrección formal en programación, que trata los algoritmos y programas como entidades matemáticas que pueden demostrarse correctas o incorrectas de forma rigurosa. Aunque este enfoque no es necesariamente habitual en la ingeniería del software, establece su importancia fundamental. Uno de los personajes clave que estudió y abogó por la programación como ciencia exacta fue Dijkstra (referencias notables son el artículo (Dijkstra 1972) y el libro (Dijkstra 1976)).∘

32: *La simplicidad* es una gran virtud
- pero requiere mucho trabajo para conseguirla
- y educación para apreciarla.
- Y para empeorar las cosas: la complejidad se vende mejor.


Edsger W. Dijkstra

∘La Crisis del Software de 1968 (un término acuñado en la Conferencia de Ingeniería del Software de la OTAN, Garmisch, Alemania, 1968) estableció que era necesario tratar el desarrollo de software como una disciplina de ingeniería propia (con especificación, diseño, pruebas y metodología) en lugar de como un oficio artesanal, y dio lugar al movimiento de la programación estructurada.

1·4.4.1 Razonamiento con Invariantes de Bucle

La corrección formal suena abstracta hasta que se ve aplicada a algo tan ordinario como un bucle. Un **invariante de bucle** es una propiedad que es verdadera antes de que el bucle comience, se mantiene verdadera tras cada iteración individual y — junto con la condición que finalmente detiene el bucle — nos permite deducir exactamente qué calcula el bucle, sin seguirlo mentalmente iteración por iteración.

Tomemos el algoritmo para sumar los primeros `n` números naturales:

1. Establecer `sum` a `0` e `i` a `1`.
2. Mientras `i` sea menor o igual que `n`: sumar `i` a `sum`, luego incrementar `i` en uno.
3. Devolver `sum`.

Como invariante, propongamos: *antes de cada iteración, `sum` es igual a `1 + 2 + ... + (i-1)`* — la suma de todo lo añadido hasta ahora. Demostrar la corrección de un bucle con un invariante siempre sigue el mismo patrón de tres pasos:

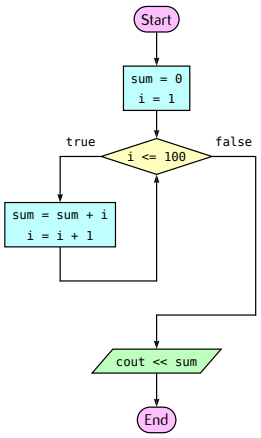
Inicialización Antes de la primera iteración, `i` vale `1` y `sum` vale `0`. La suma «de 1 a 0» está vacía, es decir, cero, por lo que el invariante se cumple trivialmente en el momento en que entramos al bucle.

Mantenimiento Supongamos que el invariante se cumple al inicio de alguna iteración, de modo que `sum` equivale a `1 + ... + (i-1)`. El cuerpo suma `i` a `sum`, dando `1 + ... + (i-1) + i`, que es `1 + ... + i`; luego incrementa `i` a `i+1`. El nuevo valor de `sum` es por tanto exactamente `1 + ... + ((i+1)-1)` — el invariante se cumple de nuevo, listo para la siguiente iteración.

Terminación El bucle se detiene en cuanto `i` supera `n`, es decir, cuando `i` es igual a `n+1` (ya que `i` solo crece de uno en uno). Sustituyendo `i =`

`n+1` en el invariante se obtiene `sum = 1 + ... + n` — precisamente el valor que el algoritmo debía calcular.

33: Diagrama de Flujo de Control del programa en Sección 2·1.2



Nótese lo que logra este argumento: es válido para **todo** valor de `n` a la vez, sin ejecutar el algoritmo ni una sola vez. Este es el tipo de razonamiento que Dijkstra tenía en mente cuando insistía en que un programa debe **probarse** correcto en lugar de limitarse a observar que funciona en un puñado de casos de prueba — y es exactamente el razonamiento que retomaremos, aplicado directamente a código C++, cuando estudiemos el diseño de bucles más adelante en el curso.

1·4.4.2 Programar con estilo

Al programar, la etapa inicial consiste en escribir el código fuente del programa. La elección del editor es bastante importante en esta etapa. Todos los editores modernos proporcionan *resaltado sintáctico*, lo que facilita enormemente la escritura de código: destacan diferentes elementos sintácticos con colores distintos, haciendo el programa considerablemente más fácil de leer.

Algunos programadores principiantes subestiman la importancia del formato estético — como el espaciado alrededor de los elementos, la organización de los bloques de código, la indentación y la estructura general del texto — pero esto se vuelve cada vez más evidente a medida que crece la longitud del programa. Además, se vuelve crucial a medida que aumenta la complejidad del programa, haciendo más difícil seguir la lógica sin un formato adecuado.

34: Indentación

```
int main() {
  int sum = 0;
  int i = 1;

  while (i <= 100) {
    sum = sum + i;
    i = i + 1;
  }

  cout << "The sum fro

  return 0;
}
```

Un área común donde los programadores principiantes encuentran dificultades es la indentación de sentencias y bloques de sentencias. Los programas consisten en grupos anidados de sentencias, y la indentación hace explícitas estas agrupaciones. Sin una indentación adecuada, los programas se vuelven ilegibles para las personas. En Figura 34, las líneas rojas verticales indican los dos niveles de indentación presentes en ese programa. Típicamente, indentamos dos espacios para avanzar al siguiente nivel dentro de un bloque, e incrementamos consistentemente el nivel de indentación para los bloques de código delimitados por llaves `{..}`.

Una buena referencia introductoria sobre estilo de escritura es el estilo de indentación K&R.

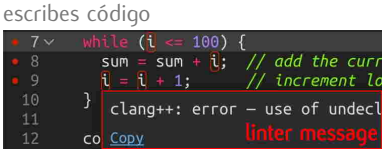
Para nuestro curso, ver Sección Part 7.

1·4.4.3 Programar con la ayuda de linters

Los linters son programas que verifican la corrección del código tanto a nivel sintáctico como, en cierta medida, semántico. Identifican posibles problemas como variables sin usar o definidas incorrectamente. Es importante destacar que realizan este análisis en el momento óptimo: mientras estás editando y compilando tu código fuente, mucho antes de encontrar errores en tiempo de ejecución. En los editores modernos, los linters se integran en el entorno de programación para el lenguaje elegido, típicamente mediante una función incorporada o un complemento.

Las herramientas más eficaces minimizan la intrusión y la distracción, preservando así tu concentración y manteniendo el flujo de desarrollo. De esta manera, los editores con linters para C++ muestran indicadores sutiles para errores comunes — como variables no declaradas y paréntesis sin cerrar — destacando errores típicos de codificación en lugar de prescribir soluciones. En Figura 35, omitimos declarar la variable `int i`. El linter entonces destaca cada ocurrencia con un rectángulo rojo mientras escribimos el código, y al pasar el ratón por encima, muestra una explicación del error. Esto ocurre mientras estás editando, incluso con el fichero sin guardar y sin la intervención del compilador, pero resulta invaluable para identificar erratas típicas, variables olvidadas y muchos otros detalles.

35: Los linters son especialmente útiles cuando se integran en los editores de texto de programación, ya que abordan omisiones y errores comunes mientras escribes código



La idea detrás de los linters no es hacer el trabajo por ti, sino ayudarte a gestionar los muchos detalles de tus programas manteniendo tu creatividad sin impedimentos.

1·5 Lenguajes de Programación

Los ordenadores solo pueden ejecutar código máquina binario, mientras que las personas dependemos de formas de representación considerablemente más ricas. Los lenguajes de programación sirven de puente entre estos dos dominios: son suficientemente formales y precisos para que las máquinas los ejecuten sin ambigüedad, y al mismo tiempo resultan legibles para las personas. Nuestro objetivo es aprender las estructuras fundamentales de programación, los tipos de datos básicos y los enfoques esenciales para la organización del código. Estos conceptos están bien cubiertos en uno de los lenguajes más clásicos y ampliamente utilizados: C y su versión ampliada C++. Este lenguaje en particular, su sintaxis y gramática constituyen el foco de este curso.



37: Nivel de abstracción del lenguaje, desde el concepto hasta la máquina



Los lenguajes de programación son independientes del sistema operativo en el que se instalan y son portables entre diferentes sistemas, gracias a los compiladores e intérpretes disponibles en esos sistemas operativos. Estas herramientas convierten la sintaxis estándar en el código binario específico requerido por cada plataforma.

Los lenguajes de programación son necesarios porque la programación de bajo nivel — es decir, programar usando instrucciones de dispositivos hardware — aunque produce el código ejecutable más rápido, es engorrosa de modificar y entender, y ofrece portabilidad limitada entre plataformas. El lenguaje natural, en cambio, es necesario para expresar la variedad mucho más amplia de significados que nos comunicamos entre las personas. Este significado se completa con el lenguaje visual, los gestos corporales y el contexto compartido construido entre los hablantes a través de su historia de comunicación. Las personas pueden comprenderse a veces con pocas o ninguna palabra, dependiendo enteramente de contextos que emergen espontáneamente y que hemos aprendido a interpretar.

Curiosamente, los grandes modelos de lenguaje son capaces de seguir los patrones que usamos y se están volviendo capaces de comunicarse con nosotros y ayudarnos cuando expresamos conceptos y problemas de forma inequívoca, de modo que la solución sea reconocible y la encontremos válida. El problema con los grandes modelos de lenguaje es que los usuarios tienden a confiar en las respuestas del ordenador como si

fueran producidas por personas reales que genuinamente comprenden las condiciones no expresadas del problema, y las personas tienden a tratar las respuestas de los grandes modelos de lenguaje como inteligentes, creyendo que poseen el contexto, la historia y el conocimiento genuino de nosotros y de lo que «dicen».

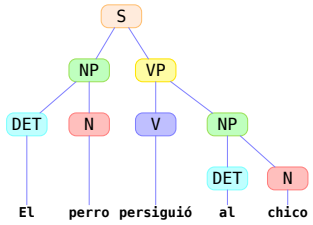
Los lenguajes de programación son fiables; las conversaciones y respuestas de los grandes modelos de lenguaje no lo son.

Todo depende de la precisión de las expresiones. Muchos programadores hoy en día creen que el futuro de los lenguajes de programación reside en los grandes modelos de lenguaje — que lenguajes como C++, Python y JavaScript son complicados, mientras que las expresiones naturales que podemos usar para consultar los generadores de lenguaje son más simples, más alineadas con nuestro propio pensamiento y, por tanto, más rápidas de articular y de producir. Este razonamiento tiene cierto mérito, pero conlleva una advertencia importante. El código generado por grandes modelos de lenguaje se basa en respuestas genéricas adaptadas a lo que los programadores han expresado en sus indicaciones. Cuanto más ambiguas son las indicaciones, menos precisas son las respuestas generadas. Los modelos deben inferir lo que la indicación omitió y aplicar soluciones estándar allí donde el programador careció de especificidad. En definitiva, aunque tal código se produce rápidamente, requiere una revisión exhaustiva y es probable que esté muy por debajo de la calidad de un programa cuidadosamente elaborado por personas que genuinamente entendían lo que querían lograr. Además, lo que las personas realmente quieren cambia con el tiempo, y el tiempo adicional que invierten en producir el código mejora su calidad en consecuencia.

1·5.1 Reconocimiento de Lenguajes. Gramáticas

Así como los lenguajes naturales están gobernados por reglas gramaticales, los lenguajes de programación están definidos por gramáticas formales que determinan qué secuencias de texto constituyen programas válidos. Por ejemplo, «Soy humano» es gramaticalmente correcto, mientras que «Humano soy yo» no lo es en el mismo sentido. La gramática de un lenguaje especifica qué elementos léxicos — letras y símbolos individuales — son admisibles, qué combinaciones de estos elementos forman palabras significativas en el vocabulario del lenguaje, y cómo pueden estructurarse estos símbolos para transmitir significado.

38: Las gramáticas definen las secuencias correctas de texto. Todos los lenguajes de programación tienen una gramática



39: Noam Chomsky en 1977, ahora con 97 años



Razonar sobre si un programa es correcto solo tiene sentido una vez que ya podemos decir, sin ambigüedad, si es un programa en absoluto — es decir, si pertenece al lenguaje definido por las reglas de C++. Esta pregunta más básica la responde una **gramática**: un conjunto finito de reglas de reescritura que genera cada cadena válida del lenguaje y ninguna más. Noam Chomsky formalizó esta idea en la década de 1950 mientras trabajaba en el lenguaje natural◆, y es ahora la herramienta estándar para definir la sintaxis de cualquier lenguaje de programación, incluido C++.

◆ La motivación original de Chomsky era lingüística, no computacional: quería una explicación matemática precisa de qué oraciones son gramaticales en un lenguaje natural como el inglés. La misma maquinaria resultó ser exactamente lo que los informáticos necesitaban pocos años después para definir los lenguajes de programación, y su clasificación de las gramáticas por poder expresivo — la *jerarquía de Chomsky* — es aún la que se utiliza hoy. Para un breve adelanto de la idea directamente de la fuente, ver el breve Capítulo 2 de (Chomsky 1957).

Formalmente, una gramática es una cuádrupla:

$$G = (N, T, I, R)$$

No terminales (N) símbolos marcadores que representan una categoría sintáctica aún por expandir (escritos aquí en mayúsculas, como P).

Terminales (T) los símbolos reales del alfabeto que pueden aparecer en una cadena terminada y válida — letras, dígitos, puntuación — que no pueden expandirse más.

No terminal inicial (I) el único no terminal, $I \in N$, desde el que debe comenzar toda derivación.

Reglas (R) reglas de reescritura de la forma *no terminal* $\to$ *secuencia de terminales y/o no terminales*, cada una especificando una forma en que un no terminal puede expandirse.

Cada conjunto de reglas se lee como «P puede reescribirse como...». Aplicando las reglas repetidamente, comenzando desde P, se genera cada cadena válida del lenguaje.

Ejemplos:

$$\begin{aligned} G_3 &: \quad P ::= z \\ G_1 &: \quad P ::= aPb \mid \varepsilon \\ G_2 &: \quad P ::= aPbb \mid \varepsilon \\ G_4 &: \quad P ::= aPbP \mid \varepsilon \end{aligned}$$

G_3: sin recursión. Hay una sola regla y no contiene ningún P, por lo que no queda nada por expandir: $L(G_3) = \{z\}$. La recursión, no el alfabeto, es lo que hace infinito un lenguaje.

G_1: un bloque anidado. Expandiendo dos veces y luego eligiendo ε: $P \to aPb \to aaPbb \to aabb$. Por tanto $L(G_1) = \{a^n b^n : n \geq 0\}$. Genera `ab`, `aabb`, `aaabbb`, pero **no** `abab`: cada a es cerrada por una b colocada justo después del bloque interior, por lo que los bloques solo pueden anidarse, nunca colocarse uno al lado del otro.

G_2: desequilibrada. Misma idea, pero cada paso añade dos símbolos de cierre: $L(G_2) = \{a^n b^{2n} : n \geq 0\}$.

G_4: paréntesis equilibrados. El segundo P después de la b permite añadir otro bloque. Leyendo a como `(` y b como `)`, esto genera todas las cadenas equilibradas:

$$P \to aPbP \to abP \to abaPbP \to abab$$

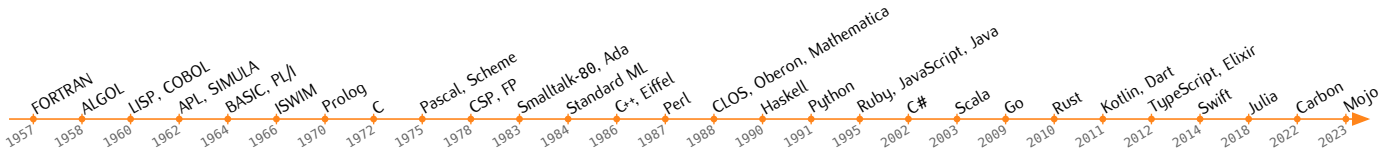
es decir, `()()`. El anidamiento también funciona: $P \to aPbP \to aaPbPbP \xrightarrow{*} aabb$, es decir, `(())`.

Cadena	G_1	G_2	G_4	En paréntesis
ab	sí	no	sí	( )
abb	no	sí	no	—
aabb	sí	no	sí	(( ))
abab	no	no	sí	( ) ( )
aabbab	no	no	sí	(( )) ( )

Solo G_4 describe realmente los paréntesis equilibrados, y es el tipo de regla que modela la estructura anidada **y** secuencial en C++ (bloques dentro de bloques, y sentencias una tras otra).

Un analizador sintáctico — parte del compilador, como veremos en breve — realiza el trabajo inverso: dada una cadena, decide si alguna secuencia de aplicaciones de reglas podría haberla producido, y rechaza la cadena (un **error sintáctico**) si no es así. Este es el primer y más económico tipo de corrección que un programa puede fallar: antes de que se considere nada sobre su lógica, debe pertenecer primero al lenguaje que define la gramática.

1·5.2 Historia de los lenguajes de programación



Las dos primeras décadas establecieron las dos familias principales que siguen en uso hoy. **FORTRAN** (1957) y **ALGOL** (1958) introdujeron el estilo imperativo y procedural que heredaron **C**, **Pascal** y sus descendientes. **LISP** (1960), en cambio, trataba los programas como datos y la computación como aplicación de funciones — el fundamento de todos los lenguajes funcionales desde **Scheme** hasta **Haskell**. Estos representan dos enfoques al mismo concepto fundamental: ambos instruyen al ordenador qué hacer paso a paso, pero divergieron pronto. El dialecto procedural imperativo emplea *haz esto, luego esto*, mientras que el dialecto funcional imperativo emplea *esto equivale a aquello*.

Desde los años setenta hasta los noventa, los lenguajes de programación evolucionaron para abordar problemas específicos en lugar de encarnar filosofías generales. C (1972) se estableció como una capa de abstracción portable cercana a la máquina; **Smalltalk** y **C++** unificaron estado y comportamiento en objetos◇; **Prolog** y **Standard ML** investigaron la lógica y el tipado fuerte ; **Perl**, **Python** y **Ruby** satisficieron la necesidad de integración rápida de sistemas. Cada nuevo lenguaje representaba menos un cambio de paradigma que una respuesta pragmática a las limitaciones percibidas en su predecesor. Perl, que sigue en uso activo hoy en día, abordó brillantemente la necesidad de los administradores de sistemas de automatizar muchas tareas pequeñas y dispares desde la línea de comandos, inicialmente con un enfoque procedural y añadiendo posteriormente capacidades orientadas a objetos. Python ofrecía un mayor nivel de abstracción, una gestión superior de bibliotecas y una sintaxis significativamente simplificada, aunque algunos argumentan que la simplificación fue demasiado lejos — por ejemplo, solo proporciona la construcción de bucle `while` y carece de una estructura de selección `switch`. Ruby, que luego destacó en el desarrollo de aplicaciones web, se distinguió por su diseño basado en objetos que iba más allá de la orientación a objetos convencional, proporcionando un rico conjunto de características de programación.

◇ Los objetos encapsulan tanto datos como comportamiento y, de forma crucial, permiten la creación de objetos descendientes que heredan esas propiedades y comportamiento. Este mecanismo de herencia es fundamental para cómo los sistemas operativos modernos permiten el desarrollo de aplicaciones complejas: proporcionan clases de objetos que las aplicaciones pueden usar como progenitores de sus elementos constitutivos, que coexisten dentro del entorno complejo más amplio.

El tipado fuerte significa que cada valor tiene un tipo fijo y bien definido que el compilador verifica antes de que el programa se ejecute, descartando toda una clase de errores — como tratar una cadena como un

número — que de otro modo solo surgirían en tiempo de ejecución. Standard ML llevó esto más lejos con la inferencia de tipos (el algoritmo de Hindley–Milner): el compilador deduce automáticamente el tipo de cada expresión, por lo que el programador obtiene la seguridad del tipado explícito sin tener que escribir una anotación de tipo en cada línea.

Desde el año 2000, el desarrollo de lenguajes ha sido impulsado principalmente por dos presiones: escalabilidad y seguridad. **Java** y **C#** introdujeron la gestión automática de memoria en el software empresarial, mientras que **Go** y **Rust** abordaron la necesidad de programación de sistemas sin los frecuentes errores de memoria que los programadores introducen en los programas en C■. **TypeScript** incorporó tipos estáticos a **JavaScript**□ una vez que las aplicaciones web superaron el scripting ligero. Los participantes más recientes, **Carbon** y **Mojo**, aún están estableciendo su posición — Carbon como sucesor experimental de C++, y Mojo como lenguaje compatible con Python diseñado para cargas de trabajo de IA — un recordatorio de que este panorama continúa evolucionando.

■ La recolección de basura es necesaria pero puede ser un quebradero de cabeza: libera al programador del seguimiento manual de cada asignación, eliminando categorías enteras de errores como los punteros colgantes y las fugas de memoria, pero los barridos periódicos del recolector introducen pausas impredecibles que pueden hacer que los lenguajes basados en GC como Go no sean adecuados para sistemas de tiempo real estricto. Rust evita este compromiso por completo con su modelo de propiedad y préstamo, aplicado en tiempo de compilación, que garantiza la seguridad de memoria sin ningún recolector en tiempo de ejecución ni sus pausas asociadas.

□ JavaScript es el lenguaje interactivo, el lenguaje que interactúa con el usuario, embebido en las páginas de aplicaciones web que las hace interactivas — originalmente diseñado en diez días en 1995 para añadir scripting simple al navegador Netscape Navigator. Su naturaleza dinámica y de tipado débil lo hacía fácil de aprender pero inadecuado para las bases de código grandes y de larga duración en que se convirtieron las aplicaciones web modernas; TypeScript aborda esto añadiendo un sistema de tipos estático opcional, verificado en tiempo de compilación y luego eliminado, por lo que es JavaScript ordinario lo que realmente se ejecuta en el navegador.

1·5.3 Traductores, Compiladores e Intérpretes

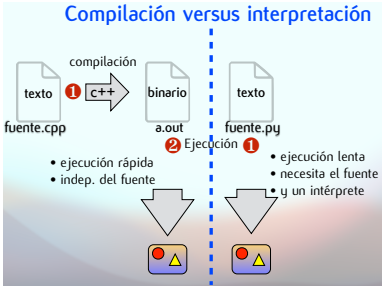
Debe existir un mecanismo que salve la brecha entre el código fuente y el código máquina. Los compiladores traducen un programa completo por adelantado, mientras que los intérpretes traducen y ejecutan código de forma dinámica — análogo a traducir un libro versus interpretar un discurso en directo. Los compiladores e intérpretes son sistemas sofisticados; siguen una compleja secuencia de pasos durante la traducción, y pueden detectarse errores en cada etapa. Estos errores pueden clasificarse como léxicos, sintácticos o semánticos. Además, incluso después de que un programa traducido (ya sea compilado o interpretado) haya superado con éxito todas estas etapas, puede contener aún errores lógicos que causen un comportamiento incorrecto durante la ejecución.

Traductor un convertidor general que traduce de un lenguaje a otro

Compilador convierte de lenguaje de alto nivel a lenguaje de bajo nivel; es específico para lenguajes informáticos. Convierte nuestro `source.cpp` en un fichero ejecutable en lenguaje máquina `a.out` que puede ejecutarse repetidamente

Intérprete opera in situ ejecutando código de alto nivel sin conversión intermedia; a medida que lee el código fuente, lo ejecuta directamente, sin pasos adicionales. Por ejemplo: Python, JavaScript o Ruby son lenguajes interpretados.

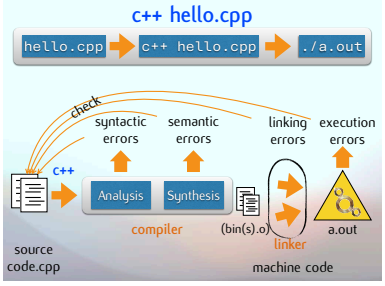
40: Los compiladores generan un nuevo fichero ejecutable independiente, mientras que los intérpretes ejecutan el código a medida que leen el código fuente original, del que dependen



Desde una perspectiva de alto nivel, el compilador realiza las siguientes operaciones sobre el fichero fuente original, varios ficheros intermedios y los componentes de la biblioteca estándar (los ficheros intermedios, denominados `name.o`, son típicamente eliminados inmediatamente por el compilador y no son visibles en nuestras sesiones):

1. Analizar el fichero fuente `source.cpp`
2. Generar código binario a partir de él, produciendo `bin.o` (pueden estar involucrados múltiples ficheros fuente y binarios `.o`)
3. Enlazar todos los ficheros `.o` junto con los componentes de la biblioteca estándar necesarios para producir un fichero ejecutable `a.out`

41: Ficheros involucrados en un proceso de compilación

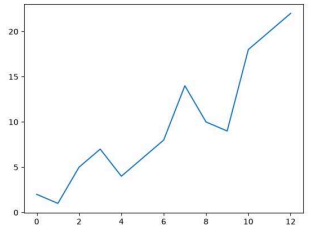


¿Cuáles son las ventajas de los intérpretes? Hay varias significativas. En primer lugar, obtienes resultados y ejecutas código inmediatamente, lo que los hace particularmente adecuados para probar soluciones, explorar diferentes enfoques y trabajar con código que requiere modificaciones frecuentes. La segunda gran ventaja de los intérpretes es que típicamente proporcionan extensas bibliotecas que ofrecen soluciones concisas a problemas comunes. Por ejemplo, analizar una página web completa en Ruby usa `nokogiri` con `doc = Nokogiri::HTML(URI.open(url))`, o en Python, representar gráficamente un array con `numpy`:

```python
# Python
import numpy as np
import matplotlib.pyplot as plt
plt.plot(np.array([2, 1, 5, 7, 4, 6, 8, 14, 10, 9, 18, 20, 22]))
plt.show()
```

Puedes observar cómo se abre la ventana gráfica en tu escritorio mientras ejecutas este código: Figura 42, y luego modificar o añadir cualquier valor y hacer clic para actualizar la salida.

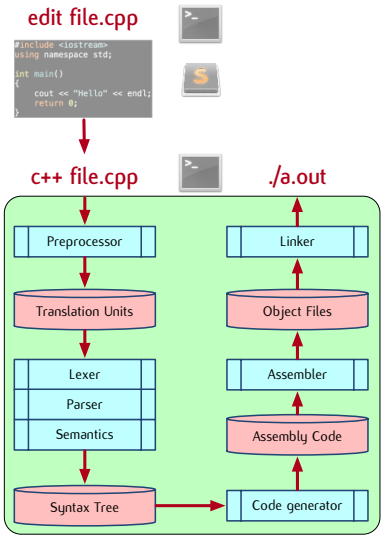
42: Salida directa del intérprete `matplotlib` de Python



1·5.3.1 Preprocesador

Antes de que el compilador propiamente dicho examine tu código, un programa separado y mucho más simple llamado **preprocesador** realiza un pase puramente textual sobre él. No entiende nada sobre la gramática, los tipos ni el alcance de C++ — solo sigue un puñado de directivas, cada una comenzando con #, y reescribe el texto en consecuencia. Su salida, no tu fichero original, es lo que el resto del proceso compila realmente: este texto reescrito se denomina **unidad de traducción** (ver Figura 43).

43: Fases de la compilación (y enlazado) desde tu fichero fuente `file.cpp` hasta la producción del `a.out` ejecutable independiente final



La directiva que ya has usado en cada programa es `#include <iostream>`: el preprocesador localiza la cabecera indicada y pega todo su contenido en lugar de esa línea, textualmente, antes de que comience la compilación — de modo que `cout` y `endl` solo son conocidos por el compilador porque sus declaraciones fueron copiadas físicamente en tu fichero momentos antes. `#define` funciona de forma similar pero para nombres de tu propia elección: `#define MAX 100` reemplaza cada ocurrencia posterior del token `MAX` por `100`, textualmente, sin ninguna noción de tipo ni alcance. Una alternativa más disciplinada, idiomática en C++ y la que preferiremos a lo largo de este curso, es una declaración `const int MAX = 100;` correctamente tipada — gestionada por el compilador, no por el preprocesador, y por tanto sujeta a las mismas verificaciones de tipo que todo lo demás.

Dado que la sustitución ocurre ciegamente, antes de cualquier comprobación sintáctica o semántica, los errores del preprocesador se manifiestan de formas inusuales: una cabecera incluida que no existe falla inmediatamente, sin referencia alguna a lo que tu programa intentaba realmente hacer con ella, mientras que un `#define` descuidado puede corromper silenciosamente código escrito muy lejos de la propia definición. Ten en cuenta esta etapa cada vez que un mensaje de error apunte a una línea que nunca escribiste, dentro de un fichero que nunca abriste — el preprocesador es muy probablemente la causa.

1·5.3.2 Enlazador

En nuestro compilador de C++, existe una etapa final en la que el compilador, tras compilar cada fichero fuente `.cpp` que compone el programa,

puede emitir un *error de enlazado*. Estos errores ocurren únicamente cuando algo es invocado pero finalmente no puede ser encontrado por el enlazador del compilador. Aunque en nuestro curso trabajaremos exclusivamente con programas que contienen un único fichero fuente `.cpp` (mientras que las aplicaciones comerciales típicamente comprenden decenas o cientos de ficheros `.cpp`), los errores de enlazado pueden ocurrir igualmente si referencias entidades que no existen o las invocas con pequeños errores sintácticos. Por ejemplo, si cometes el error de llamar a la función $\sqrt{x}$ (en C++ `sqrt(x)`) pero escribes `sQrt()` en su lugar, el compilador no detectaría ningún error sintáctico. Este error pasa a la fase final, el enlazador, sin ninguna referencia a la línea en que ocurrió en el fichero fuente `myProgram.cpp`. El enlazador buscará la función `sQrt()` en todas las bibliotecas estándar sin éxito y emitirá inevitablemente un mensaje de error indicando que `sQrt()` no fue encontrada, sin referencia a la línea en que ocurrió el error.

1·5.3.3 Depuración

La depuración es el proceso de localizar errores — fallos que se han colado en la lógica de tu programa. Tu programa puede compilar y ejecutarse, pero fallar ruidosamente durante la ejecución o producir resultados incorrectos. En ese momento, tu único recurso es reescribir el código, pero identificar qué parte está fallando puede ser difícil. Cuando los programas constan de múltiples ficheros, este problema se vuelve aún más complicado. Afortunadamente, existen programas especializados llamados *depuradores* para ayudar. Un depurador puede ejecutar tu programa paso a paso, mostrando la salida junto al código fuente que la produce. Puedes insertar puntos de interrupción en los que la ejecución se detiene, permitiéndote inspeccionar el estado actual (valores de variables y salida hasta ese momento), y típicamente proporcionan herramientas para examinar todo lo que tu programa gestiona. Sin embargo, usar un depurador puede ser como usar una maza para partir una nuez. Especialmente para programas cortos como los nuestros, es mucho más práctico insertar simples sentencias de impresión que muestren los valores de las variables, exteriorizando así lo que ocurre dentro del programa. Si el programa falla, estas sentencias también revelan cuándo ocurrió el fallo. Por ejemplo, un simple

44: La depuración implica someter nuestro programa a escrutinio trazando su ejecución línea por línea o deteniéndose en puntos de interrupción, mientras se observan simultáneamente los cambios en las variables del programa. Sin embargo, para programas simples, esto puede ser un proceso excesivamente meticuloso




```
r = a / b;
```

puede hacer que nuestro código falle si `b` es cero. Para detectar que esto ocurre, puedes añadir:

```
cout << "before division. a: " << a << ", " << "b: " << b << endl;
r = a / b;
```

Esto mostrará el valor de las variables `a` y `b` justo antes de que el programa falle, tras lo cual sale sin continuar.

1·6 Visión General de un Sistema Informático

El hardware solo es inerte: un procesador, un disco o una pantalla no hacen nada por sí mismos. Un **sistema** informático solo cobra vida cuando el hardware se combina con software organizado en capas, cada capa apoyada en los servicios proporcionados por la que tiene por debajo — como un edificio que descansa sobre sus cimientos y se eleva piso a piso.

Sistema operativo La capa permanente y más baja: gestiona el hardware directamente — planificación de la CPU, memoria, ficheros, dispositivos — y ofrece a todos los demás programas una plataforma común sobre la que ejecutarse. Ejemplos: Windows, macOS, Linux

Software de sistema y desarrollo Construye y mantiene otro software: compiladores, enlazadores, depuradores, sistemas de control de versiones — los traductores discutidos anteriormente pertenecen a esta categoría

Utilidades Herramientas auxiliares que mantienen el sistema funcionando con fluidez pero que no producen ellas mismas documentos para el usuario: gestores de ficheros, software antivirus, herramientas de limpieza del disco

Software de aplicación Produce o manipula documentos y resultados para el usuario final: procesadores de texto, navegadores web

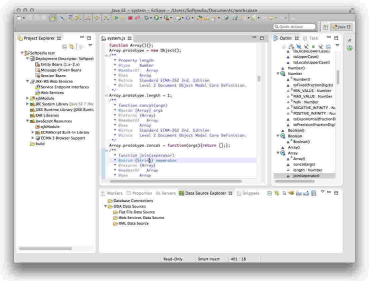
45: Capas de un ordenador



1·6.1 Entorno de Desarrollo Integrado

Los compiladores, o más en general los Entornos de Desarrollo Integrados (IDEs), se sitúan en la capa de sistema y desarrollo: son las herramientas de software responsables de construir el software de aplicación que estamos a punto de escribir. En este curso, debemos instalar primero un compilador en nuestros ordenadores personales para familiarizarnos con él y trabajar eficientemente. Esta tarea de configuración puede ser complicada para un curso introductorio, por lo que es importante que la completes lo antes posible. No dudes en pedir ayuda a los profesores y compañeros si es necesario.

46: IDE Eclipse. Típicamente ficheros en la columna izquierda, ejecución y registros en la parte inferior, jerarquía de objetos a la derecha. Útil para construir aplicaciones complejas que involucran un conjunto de ficheros y bibliotecas



Un IDE integra un editor, un compilador y un terminal en un entorno de desarrollo unificado — el equivalente para el programador de un taller bien equipado. Es la herramienta que utilizarás a lo largo de las sesiones de laboratorio. Para nuestros propósitos, los IDEs proporcionan los tres componentes esenciales necesarios para construir, compilar y probar la ejecución de cada programa en este curso:

1. Editor
2. Compilador
3. Terminal para la ejecución

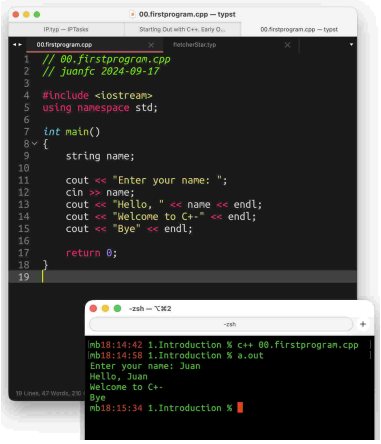
47: Iconos para representar una ventana de terminal (Intérprete de Línea de Comandos o CLI)



1·6.2 Entorno de Desarrollo No Integrado

En la práctica, solo necesitas una línea de comandos UNIX/Linux, una ventana de terminal y un editor de texto para editar, compilar y ejecutar los programas de este curso. De hecho, esta es la configuración preferida para muchos — yo mismo incluido —, ya que no distrae y mantiene el foco en lo que importa. Además, requiere significativamente menos recursos del sistema y hace transparente cada paso del proceso de compilación.

48: Usar solo un buen editor de texto simple o un editor de texto de programación y una ventana de terminal (CLI) hace las cosas considerablemente más fáciles y adecuadas para este curso



El sistema debe tener un compilador de C++. Esto es sencillo de configurar en Linux, macOS y Windows (con Linux virtualizado).

Todo esto se explicará con detalle durante las sesiones de laboratorio.

Puedes editar, compilar y ejecutar código directamente sin depender de un IDE gráfico. Aunque esto puede parecer más complicado que

usar entornos gráficos, para el tipo de ejercicios que realizaremos — que típicamente no constituyen proyectos formales — las características adicionales de los IDEs gráficos a menudo introducen complejidad innecesaria en lugar de proporcionar un beneficio genuino.

1·6.3 Punto de Control

1) ¿Qué es un algoritmo?
2) ¿Por qué hay tantos lenguajes de programación, y por qué se inventaron en un primer momento?
3) ¿Cuáles son los componentes fundamentales de una gramática?
4) ¿Por qué los lenguajes de programación son independientes del sistema operativo y portables entre diferentes sistemas?
5) ¿Cuál es la distinción entre lenguajes de alto nivel y de bajo nivel?
6) ¿Qué significa portabilidad?
7) Describe las funciones que realizan el preprocesador, el compilador y el enlazador.
8) ¿Qué contiene un fichero fuente, un fichero objeto y un fichero ejecutable?
9) ¿Qué es un entorno de desarrollo integrado?
10) ¿Cuál es la configuración mínima requerida en un ordenador para poder programar?
11) ¿Qué es un error sintáctico?
12) ¿Qué es un error semántico?
13) ¿Qué es un error lógico?
14) ¿Qué es un error en tiempo de ejecución?
15) ¿Qué es el pseudocódigo?

Referencias

CHOMSKY, Noam, 1957. *Syntactic Structures*. The Hague: Mouton.

El libro que convirtió la gramática formal en un objeto matemático y, con ello, en la teoría que se utilizaría después para definir la sintaxis de los lenguajes de programación. Escrito para lingüistas, no para informáticos, y la mayor parte puede omitirse sin problema para un programador de primer año; pero el capítulo 2, «La independencia de la gramática» (unas cinco páginas), es una lectura rápida y autocontenida: muestra, con la famosa frase *Colorless green ideas sleep furiously* (Ideas verdes incoloras duermen furiosamente), que una cadena puede ser perfectamente gramatical y ser un sinsentido, o ser agramatical usando solo palabras familiares. Ese único ejemplo resume la razón de ser de una gramática formal en una sola frase, y no requiere conocimientos previos para apreciarlo. Leer solo ese capítulo.

DIJKSTRA, Edsger W., 1972. The Humble Programmer. *Communications of the ACM*. 1972. Vol. 15, no. 10, p. 859-866. DOI **10.1145/355604.361591**.

La conferencia Turing de Dijkstra de 1972 y uno de los textos más importantes en la historia de la computación. Escrito con una claridad y convicción notables, sostiene que la programación es una disciplina intelectual del más alto orden, no un arte de trucos ingeniosos, sino una actividad rigurosa que exige humildad ante la complejidad. Dijkstra recorre la crisis del software de los años 60, desestima el culto a la

línea única y sostiene que los programas deben componerse correctamente desde el principio, no depurarse hasta ser correctos a posteriori. La conferencia cierra con un llamamiento a abordar cada programa como un Programador Muy Humilde. Lectura imprescindible para los estudiantes de primer año: breve, bellamente escrita y un reto directo a los hábitos mentales que producen código malo.

DIJKSTRA, Edsger W., 1976. *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall. ISBN 0-13-215871-X.

Una obra de referencia en la que Dijkstra desarrolla una metodología formal para derivar programas correctos a partir de sus especificaciones, introduciendo las precondiciones más débiles y los transformadores de predicados. Lectura exigente, pero fundamental para comprender por qué la corrección de los programas debe razonarse y no simplemente comprobarse con pruebas. Se complementa de forma natural con el artículo *The Humble Programmer* como base filosófica del curso.

Tarea 0: Editar y compilar nuestro primer fichero fuente en C++

Estos dos ejercicios tratan de conseguir que tus primeros programas compilen y se ejecuten. No necesitas entender cada línea todavía — céntrate en el ciclo editar–compilar–ejecutar. Rellena siempre el comentario de cabecera con el nombre del fichero, tu nombre o alias, la fecha y una breve descripción de lo que hace el programa.

1 **Cuenta atrás.** El siguiente programa cuenta del `1` al `10`. Modifícalo para que cuente hacia atrás del `10` al `1`.

```C++
// 01.countTo10.cpp
// [yourShortName] [date]
#include <iostream>
using namespace std;

int main()
{
    int i = 1;
    while (i <= 10) {
        cout << i << endl;  // print the index
        i = i + 1;          // update index
    }
    return 0;
}
```

Envíalo como `01.countTo10.cpp` al `buzón de Tarea 0`.

2 Copia el siguiente programa (*no necesitas entenderlo todavía*) y añade opcionalmente un breve comentario describiendo lo que crees que hace.

```C++
// 02.nameMe.cpp
// [yourShortName] [date]
// This program [does .. description]

#include <iostream>
using namespace std;

int main()
{
    int sum = 0;
    int i = 1;

    while (i <= 100) {
        sum = sum + i;  // accumulate
        i = i + 1;      // update loop variable
    }

    cout << "The result of this is: " << sum << endl;

    return 0;
}
```

Envíalo como `02.nameMe.cpp` al `buzón de Tarea 0`.

o O o

Part 2 Introducción a un Lenguaje de Programación

Contenido de esta parte Contenido del libro

2·1 Introducción a C++ 34
2·2 Tipos de Datos Simples 37
2·3 Constantes, Variables y Asignaciones 41
2·4 Caracteres y cadenas 52
2·5 Enumeración de los valores que puede contener un tipo 54
2·6 typedef: Definición de nuevos nombres para tipos 55
2·7 Números aleatorios 57
2·8 Precedencia de operadores 57
2·9 Entrada y Salida Básicas 59
▸ Tarea 1: Fundamentos. Programas secuenciales: aritmética y E/S estándar 62
2·10 Flujo de control 64
2·11 Expresiones booleanas 64
▸ Tarea 2: Booleanos 68
2·12 Estructuras de selección 71
▸ Tarea 3: Estructuras de control. Condicionales 80
2·13 Estructuras de iteración 83
2·14 Gestión de errores y excepciones 89
2·15 Recomendaciones y errores frecuentes 91
▸ Tarea 4: Estructuras de control. Bucles simples 94
▸ Tarea 5: Estructuras de control. Bucles dobles 98

Habiendo estudiado los algoritmos como recetas abstractas, necesitamos ahora una notación para expresarlos de modo que un ordenador pueda ejecutarlos. Este capítulo presenta C++, el lenguaje que utilizaremos a lo largo del curso, cubriendo sus bloques constructivos básicos: tipos de datos, expresiones, entrada y salida, y las estructuras de control que gobiernan el orden en que se ejecutan las instrucciones.

2·1 Introducción a C++

C++ es uno de los lenguajes de programación más influyentes y ampliamente utilizados jamás creados, y potencia desde sistemas operativos hasta videojuegos. Comenzamos con su historia, un primer ejemplo completo y la forma general que sigue todo programa C++.

2·1.1 La Evolución de C++

C nació entre 1969 y 1973 como el lenguaje de elección para el desarrollo del sistema operativo UNIX. Linux = versión libre de UNIX.

C++ era originalmente «C con clases» y comenzó a desarrollarse en 1979. C++ fue creado por Bjarne Stroustrup a principios de los años 80 como una extensión del lenguaje C, introduciendo abstracciones de alto nivel mientras mantenía la eficiencia de C. Comprender sus orígenes aclara muchas de sus decisiones de diseño y algunas de sus idiosincrasias.

C++ es compatible hacia atrás con C: un compilador de C++ puede compilar código fuente C también. Existen muchos otros lenguajes basados en C: Objective-C, C#, etc.

Versión	Cambios principales
C++98	La primera versión estándar
C++03	Inicialización de valores
C++11	Expresiones lambda, constructores delegados, sintaxis de inicialización uniforme, `nullptr`, deducción automática de tipos y decltype, referencias rvalue, etc.
C++14	Lambdas polimórficas, separadores de dígitos, captura lambda generalizada, plantillas de variables, literales enteros binarios, cadenas entre comillas, etc.
C++17	Expresiones fold, literales de punto flotante hexadecimales, literales de carácter u8, sentencias de selección con inicializadores, variables inline, etc.
C++20	Capacidades extendidas para inspeccionar entidades del programa como variables, enumeraciones, clases y sus miembros, lambdas y sus capturas, etc.
C++23	`std::expected` para manejo explícito de errores, deducción de this, `std::mdspan`, `std::print`/`std::println`, `if consteval`, operador de subíndice multidimensional, `std::stacktrace`, `flat_map`/`flat_set`, rangos y vistas ampliados, etc.
C++26	Actualización próxima

Se puede usar cualquiera de las versiones recientes del compilador disponibles en la mayoría de los entornos de desarrollo integrados, y raramente se encuentran incompatibilidades, ya que la sintaxis utilizada en este curso es compatible con todas las versiones admitidas.

2·1.2 Ejemplo de un Programa C++

La forma más eficaz de aprender un lenguaje es estudiar un programa pequeño pero completo escrito en él. Examinamos un ejemplo mínimo línea a línea, identificando las construcciones que se repetirán en cada programa que escribamos.

```cpp
// sum1to100.cpp
// juanfc 2026-09-09
// This program sums all natural
// numbers from 1 to 100

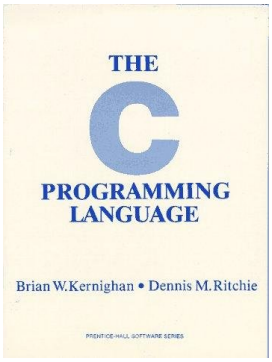
#include <iostream>
using namespace std;

int main() {
  int sum = 0;
  int i = 1;

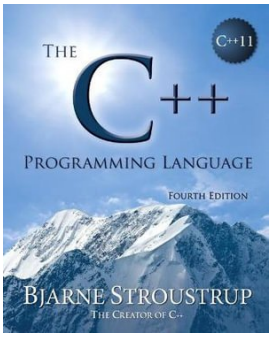
  while (i <= 100) {
    sum = sum + i;  // add the current value of i to the sum
    i = i + 1;      // increment loop variable
  }

```

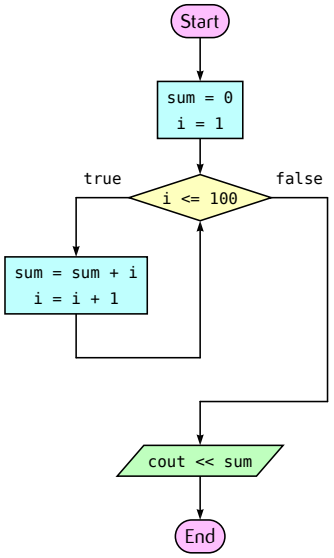
49: Libro de referencia oficial de C, 1978



50: Libro de referencia oficial de C++ para C++11



51: Flujo de control del bucle `while` que suma de 1 a 100 en Sección 2·1.2. La rama `true` vuelve a la condición; la rama `false` sale del bucle e imprime el resultado.



```
18    cout << "The sum from 1 to 100 is: " << sum << endl;
19
20    return 0;
21 }
```

2·1.3 Elementos Básicos de C++

Como cualquier lenguaje de programación, C++ comprende un conjunto reducido de componentes fundamentales: palabras clave, identificadores, literales, operadores y comentarios. Esta sección presenta estos elementos y explica las reglas que rigen su combinación.

Las palabras clave son palabras reservadas que el lenguaje reserva para su propio uso y no pueden reutilizarse con ningún otro propósito. Así, por ejemplo, nunca se puede nombrar una variable ~~`int while = 3;`~~

Flujo de control	`if`, `else`, `switch`, `case`, `default`, `for`, `while`, `do`, `break`, `continue`, `return`, `goto`
Tipos fundamentales	`void`, `char`, `int`, `float`, `double`, `long`, `short`, `signed`, `unsigned`
Almacenamiento y tipo	`const`, `static`, `extern`, `volatile`, `register`, `auto`, `typedef`
Tipos definidos por el usuario	`struct`, `union`, `enum`
Booleano y nulo	`bool`, `true`, `false`, `nullptr`
Clases	`class`, `public`, `private`, `protected`, `virtual`, `friend`, `explicit`, `mutable`, `this`, `override`, `final`
Plantillas	`template`, `typename`
Memoria	`new`, `delete`
Espacios de nombres	`namespace`, `using`
Excepciones	`try`, `catch`, `throw`, `noexcept`
Conversiones	`static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`, `typeid`
Palabras operador	`and`, `or`, `not`, `and_eq`, `or_eq`, `xor`, `xor_eq`, `not_eq`, `bitand`, `bitor`, `compl`
Tipos de carácter	`wchar_t`, `char16_t`, `char32_t`
Otros	`inline`, `operator`, `asm`, `decltype`, `constexpr`, `static_assert`, `thread_local`, `alignas`, `alignof`, `sizeof`

En el ejemplo anterior encontramos los siguientes elementos clave:

Comentarios Todo lo escrito después de `//` hasta el final de la línea es completamente ignorado por el compilador. El preprocesador intercepta y elimina estos comentarios antes de que el compilador procese el código.

Directiva de preprocesador Cualquier línea que comience con `#` seguida inmediatamente de una palabra clave constituye una directiva de preprocesador: una instrucción para el preprocesador. En este caso, `#include <iostream>` indica al preprocesador que inserte el archivo de biblioteca estándar `iostream` desde los directorios específicos del compilador. Si necesita incluir un archivo estándar, utilice `#include "nombreDelArchivo.h"` con comillas.

Declaración de espacio de nombres La sentencia `using namespace std;` permite al compilador resolver nombres de un espacio de nombres especificado sin necesitar el prefijo del espacio de nombres. Por ejemplo, esto permite usar `cout` en lugar del nombre completamente cualificado `std::cout`.

La función `main()` Es el punto de entrada del programa. Contiene las instrucciones iniciales y diversas sentencias tanto de inicialización como de ejecución. Las dos primeras líneas dentro del bloque `{....}` inicializan variables, mientras que la construcción `while (..) {` posterior implementa un bucle que ejecuta repetidamente un bloque de sentencias. Finalmente, el resultado se envía a la salida estándar mediante la sintaxis C++ `cout <<` , y el control regresa al sistema operativo con código de salida cero.

2·1.4 Estructura General de un Programa C++

Cada programa que escribamos en este curso seguirá el mismo esqueleto: inclusión de bibliotecas, declaraciones y una función `main()` que contiene las instrucciones. Fijar esta plantilla ahora nos permite concentrarnos después en lo que va dentro de ella. En el ejemplo anterior `sum1to100.cpp` y casi el resto de nuestros ejercicios, utilizaremos la plantilla:

```cpp
// 13.nameOfThisProgram.cpp
// myShortName 2026-09-09
// Optionally a short description
// of the program or notes for its use

#include <iostream>
using namespace std;

int main() {
  // instructions to execute
  return 0;
}
```

2·2 Tipos de Datos Simples

Los programas manipulan datos, y cada dato en C++ posee un **tipo** que determina qué valores puede representar y qué operaciones son válidas para él. Los tipos simples (escalares y de punto flotante▲) almacenan un único valor y sirven como bloques constructivos fundamentales a partir de los cuales se construyen todas las demás estructuras de datos.

▲Un tipo escalar es aquel en que cada valor tiene un predecesor y un sucesor bien definidos. Este concepto se aplica de forma natural a las letras y puede extenderse a los booleanos tratando `false` como menor que `true`, pero no se aplica de forma significativa a los números de punto flotante tal como se representan en los ordenadores: por ejemplo, ¿cuál es el predecesor de 3,1?

La mayoría de los lenguajes de programación no proporcionan un conjunto exhaustivo de tipos integrados para codificar todas las formas posibles de datos. Los programas deben definir estructuras más específicas y complejas para representar la información con la que trabajan, como imágenes, audio o bases de datos empresariales. Sin embargo, la mayoría de los lenguajes de programación, incluido C++, proporcionan una selección de tipos de datos simples básicos capaces de codificar enteros, números reales y caracteres.

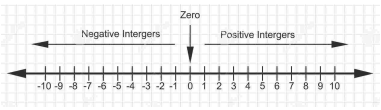
Una variable de tipo simple solo puede almacenar una única pieza de información que no puede subdividirse. Un vector físico, por ejemplo, puede almacenar tres números reales simples. Sin embargo, un número real en sí mismo se considera simple en el sentido de que es monolítico.

2·2.1 Tipos Simples Predefinidos

C++ proporciona un conjunto de tipos integrados para manejar caracteres y números (enteros y números reales):

Enteros Representan números enteros sin parte fraccionaria. Los enteros pueden ser con signo (`int`) o sin signo (`unsigned int`), y su capacidad de almacenamiento puede variar desde pequeña (`short int`) hasta grande (`long int`, `long long int`).

52: Los números enteros

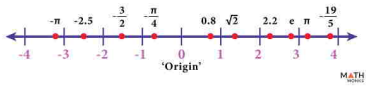


Números reales Aunque muchos programas no requieren números con parte fraccionaria, los números reales son esenciales en muchas aplicaciones. Sin embargo, son más engorrosos de manejar porque no pueden compararse para verificar igualdad de forma fiable debido a errores de precisión, y son computacionalmente más costosos de manipular. Se pueden declarar variables para números reales usando un rango menor (`float`) o un rango mayor (`double` o `long double`). Recuerde que

$$3.14 \times 10^{-3} \rightarrow \texttt{3.14E-3}$$

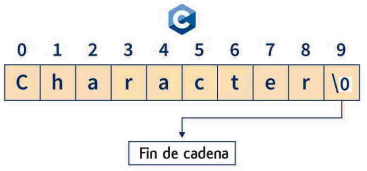
.

53: Números reales



Caracteres Las letras del alfabeto y los símbolos básicos de escritura se codifican usando la tabla ASCII estándar y las codificaciones UTF más completas, que admiten una amplia gama de caracteres específicos de idioma, símbolos matemáticos y emojis. Para el alfabeto ASCII (el único conjunto de caracteres que admite nuestra sintaxis del lenguaje), se pueden almacenar caracteres en variables de tipo `char`.

54: Caracteres, en C clásico, siempre terminados en `\0`. En C++ se guardan en el objeto `string`



Booleanos El tipo de dato más simple, utilizado para almacenar el valor de verdad de una condición, puede declararse usando el tipo de variable `bool`.

```
if (a == b)
  cout << "a is equal to b";
bool isTrue = b > 5;
```

A veces resulta conveniente escribir valores literales con dígitos separados por `'` cuando son largos, omitir el `0` inicial en `0.123`, u omitir el cero final tras el único dígito significativo en `1.`, como se muestra a continuación:

```
int n = 123'456;
float r = 3.141'592'653;
float p = .123;
float q = 1.;
```

Sin embargo, esta sintaxis solo está disponible a partir del estándar C23.

```
float r = 2 + 3.f;
```

Dado que la constante `3.14` es `double` por defecto, si se necesita un literal `float` hay que especificarlo explícitamente:

```
float r = .123f;
```

En la práctica, C++ incluye una biblioteca estándar para manejar números complejos; sin embargo, los crearemos nosotros mismos como ejercicio.

```
#include <complex>
std::complex<double> z(3, 4);    // 3 + 4i
```

Por el contrario, C++ no incluye una biblioteca estándar para manejar fracciones. Podemos importarlas desde bibliotecas externas, como

```
#include <boost/rational.hpp>
boost::rational<int> a(1, 2), b(1, 3);
auto c = a + b;   // 5/6, automatically reduced
```

o implementarlas nosotros mismos como ejercicio. Lo haremos cuando hayamos cubierto los tipos de datos estructurados.

Cada tipo tiene su propio rango y huella de memoria. Seleccionar el tipo apropiado representa la primera decisión de diseño del programador.

Codificación de:	Ejemplos de valores	Sintaxis de declaración
Booleanos	`true false`	`bool`
Caracteres	a 9 ;	`char`
Números naturales	`0 1 33`	`unsigned [long] [int]`
Números enteros	`-3 0 1000`	`[long] int`
Números reales	$3.14 \times 10^{-3} \rightarrow$ `3.14E-3`	`[long long] float/double`

Verá que el tipo de programación más frecuentemente utilizado es `int`, y es suficientemente versátil como para que `unsigned` a menudo sea sustituido por él. En consecuencia, típicamente utilizamos solo `int` y pasamos por alto la necesidad de `unsigned`△

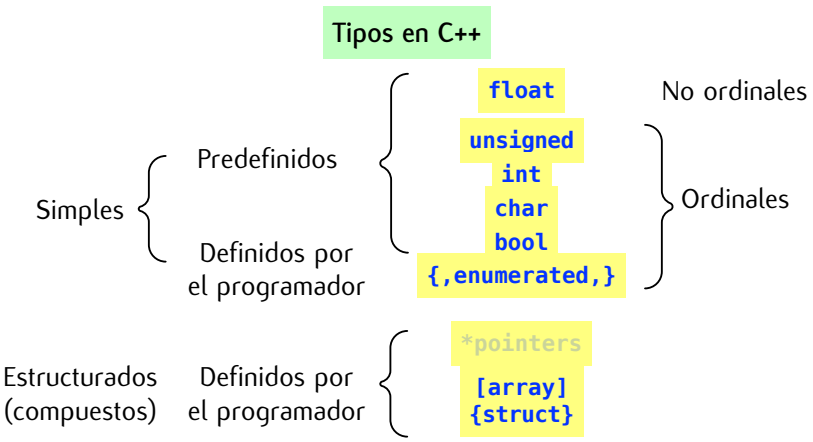
△ Recuerde que asignar un número negativo a una variable `unsigned` es muy perjudicial, ya que el resultado es completamente incorrecto.

2·2.2 Tipos Simples Definidos por el Programador

A veces los tipos predefinidos no expresan adecuadamente nuestra intención: una variable que almacena un día de la semana es mejor declararla como tal que como un entero sin procesar. Los tipos enumerados nos permiten crear nuestros propios tipos simples, haciendo los programas más claros y seguros.

Sin adentrarnos en composiciones estructuradas de múltiples tipos simples, en C++ tenemos la posibilidad de crear un nuevo tipo simple. Podemos enumerar los valores que puede contener una variable. Por ejemplo, una variable `trafficLightColor` puede contener tres valores posibles, y la mejor manera de especificarlos es por sus nombres: `GREEN`, `YELLOW` y `RED`. Nuestro lenguaje es capaz de construir esta lista de etiquetas (`GREEN`, `YELLOW` y `RED`) y garantizar que la variable `trafficLightColor` contenga una de ellas.

```
enum TLightColor {GREEN, YELLOW, RED};
TLightColor trafficLightColor = GREEN;
```



2·2.2.1 Qué implica la declaración de una variable

Cuando declaramos una variable, estamos especificando cuatro cosas importantes:

1) un nombre o identificador
2) un tipo, que determina cómo se codifica e interpreta la información
3) un tamaño, que define la capacidad de la variable y, en consecuencia, la cantidad de memoria que reserva
4) un valor inicial con el que inicializar la variable

```
int n = 10;
float r = 0.1;
```

Las variables no son simplemente contenedores de información; podemos operar con sus contenidos para producir nuevos valores, típicamente del mismo tipo que los operandos, usando operadores. Cada tipo de variable tiene su propio conjunto de operadores disponibles.

2·2.3 Operadores

Los operadores combinan valores —generalmente del mismo tipo, ocasionalmente de tipos diferentes— para formar expresiones, que pueden ser aritméticas, relacionales o lógicas, dependiendo de qué operadores se utilicen. Cada tipo de expresión hace uso de los operadores apropiados para los tipos de los valores involucrados.

Los operadores son los verbos del lenguaje. Los presentamos a continuación, junto con sus reglas de precedencia, que determinan el orden en que se evalúa una expresión compuesta.

En un lenguaje de programación, distinguimos los operadores, que actúan sobre valores, de las funciones o procedimientos que actúan sobre parámetros. Por ejemplo, `sin(x)` es la función `sin( )` que actúa sobre la variable `x` y devuelve un resultado. Del mismo modo, `x + y` utiliza el operador binario implícito `+`, que también devuelve un valor. Los operadores pueden ser unarios, como `-x`, o binarios. El único operador ternario que C++ proporciona es el **operador condicional** `?:`, que toma tres operandos: la condición antes de `?` y los dos valores alternativos. Esto lo convierte en una alternativa compacta a una sentencia `if`/`else` completa cuando se elige entre dos valores, aunque el anidamiento excesivo reduce rápidamente la legibilidad. Lo examinaremos, junto con `if`/`else`, en la sección sobre condicionales.

El único operador ternario que C++ proporciona es el **operador condicional** `?:`, y ningún otro operador del lenguaje toma tres operandos (la condición antes de `?` y las dos salidas). Esto lo convierte en una alternativa compacta a una sentencia `if`/`else` completa cuando se elige entre dos valores, aunque anidarlo demasiado profundamente perjudica rápidamente la legibilidad. Lo examinaremos, junto con `if`/`else`, en la sección sobre condicionales.

Aridad	Operadores	Tipo
Unario	`++ --`	Operador unario
Binario	`+ - * / %`	Operador aritmético
	`< <= > >= == !=`	Operador relacional
	`and or not`	Operador lógico
	`& | << >> ~ ^`	Operador a nivel de bits
	`= += -= *= /= %=`	Operador de asignación
Ternario	`?:`	Condicional

2·3 Constantes, Variables y Asignaciones

Las variables son posiciones de memoria con nombre cuyo contenido puede cambiar, mientras que las constantes almacenan valores fijos. La instrucción de asignación —colocar un valor en una posición de memoria — representa la operación más fundamental en la programación imperativa , y merece un examen más detallado del que su apariencia sencilla podría sugerir.

existe otra clase de lenguajes de programación, menos comunes en ingeniería pero más frecuentes en la investigación en informática, conocidos como programación *declarativa*. En tales lenguajes, no se especifican instrucciones para que el ordenador las ejecute; en su lugar, se declara lo que se conoce y luego se consulta el sistema con preguntas sobre el conocimiento expresado. Estos lenguajes no tienen una operación de asignación

Al declarar una variable, se puede indicar que su valor nunca será modificado. Estas son las **constantes**. Las constantes requieren que se especifique su valor inicial en el punto de declaración, y tras esta inicialización, el valor no puede cambiarse.

```
const float PI = 3.14;
const char END = 'Q';
const int MAXSIZE = 101;
```

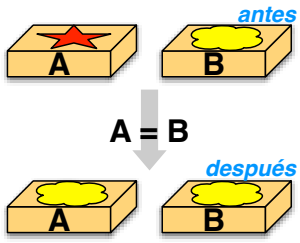
Cualquier intento posterior de asignar un nuevo valor a estas constantes hará que el compilador emita un error y detenga la compilación. ~~`PI = 3.141;`~~

Ya hemos visto cómo declarar variables (especificando su tipo, nombre identificador y habitualmente un valor inicial), y dónde declarar las variables necesarias para nuestros programas (cerca del lugar donde se usan por primera vez, para que comprendamos su significado no solo por su nombre identificador sino también por su uso real). También hemos examinado el ámbito de las variables: el rango de validez donde pueden usarse, y los bloques en los que se declaran definen el ámbito de esa declaración.

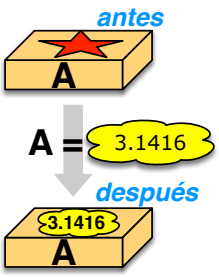
2·3.1 El operador de asignación

En los lenguajes de programación, cuando se necesita procesar información preservando el valor original, típicamente se copia el contenido de una variable a otra y luego, si es necesario, se modifica la copia mediante operaciones posteriores. Los distintos lenguajes emplean una sintaxis variada para esta operación. Algunos usan `let b = a`, interpretando el signo `=` como **copiar** el valor de la derecha en la variable de la izquierda. Otros usan `b := a`. Nuestro lenguaje, como muchos otros, simplifica esta operación de uso frecuente a: `b = a;`. Esto no indica que los valores sean equivalentes en sentido matemático; más bien significa que, tras ejecutarse la operación, el contenido de la variable `b` será igual al contenido de la variable `a`, independientemente de lo que `b` contuviese antes. Tras la operación, ambas variables almacenan contenidos idénticos en posiciones diferentes de la memoria RAM.

55: Asignación de un valor de una variable a otra variable



56: Asignación de un valor a una variable



Se pueden asignar muchos tipos de valores a una variable, pero no todos ellos. Para asignar un valor a una variable, dicho valor debe ser compatible con el tipo de la variable.

Cuando se asigna una constante a una variable, el valor literal de la constante se copia en la variable, haciendo que el contenido de la variable sea igual a ese valor. Las constantes en sí son efímeras: solo existen durante la ejecución de la operación y no tienen una posición de memoria persistente que pueda inspeccionarse o modificarse después. Las variables, por el contrario, ocupan una posición de memoria fija y

con nombre que permanece accesible a través de su identificador durante toda su vida útil, lo que es lo que permite modificarlas repetidamente a medida que el programa se ejecuta.

Solo podemos copiar cosas **hacia** variables: a la izquierda del signo `=` solo puede haber una variable.

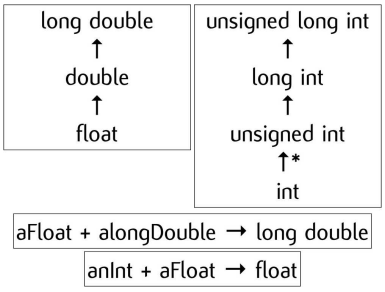
A la derecha del signo `=`, sin embargo, puede haber una constante (`a = 3;`) o una expresión que produzca un valor compatible con la variable de la izquierda (`a = 2 + 5 * b;`).

2·3.2 Conversiones de Tipo

¿Qué ocurre cuando un entero se encuentra con un número real en la misma expresión? C++ convierte valores de un tipo a otro, a veces de forma silenciosa y a veces a nuestra petición. Comprender estas conversiones evita errores sutiles y frecuentes.

A menudo, el destino de la izquierda y el valor de la derecha son simplemente incompatibles, y el compilador emite un error: `aLetter = 3.1416;` no tiene sentido para el compilador.

57: Conversiones de tipo implícitas que C++ realiza automáticamente. Algunas de ellas (`int` ⟶ `unsigned`) son particularmente peligrosas



En otros casos, aunque los tipos difieran, el compilador realiza automáticamente una conversión de su propia elección —a menudo sensata, pero no necesariamente la que se pretendía—. Por ejemplo, el compilador de C++ puede convertir un entero sin signo a un entero con signo con razonable seguridad, pero lo contrario —convertir un entero con signo a un entero sin signo— puede ser problemático: `unsigned int a = -3;` almacena `4294967293` en la variable `a`. Del mismo modo, `int a = 3.9999;` almacena `3` en `a`, truncando en lugar de redondear la parte fraccionaria.

En casos prácticos

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

 i = i  +  c;  // c -> int
 i = i  +  s;  // s -> int
 u = u  +  i;  // i -> unsigned int
 l = l  +  u;  // u -> long int
ul = ul +  l;  // l -> unsigned long int
 f = f  + ul;  // ul -> float
 d = d  +  f;  // f -> double
ld = ld +  d;  // d -> long double
```

2·3.2.1 Casting o conversiones explícitas

Podemos leer el contenido de una variable convirtiéndolo a otro tipo, tratándolo como si se asignara a ese tipo. De esta manera, podemos escribir

`int(aRealVariable)` para convertir el contenido de `aRealVariable` a un entero antes de realizar operaciones adicionales.

Un ejemplo importante es: `float r = float(1)/3;`. En este caso, sin casting explícito, `float r = 1/3;` asignaría `0` a la variable `r`, ya que la división se realiza sobre operandos enteros (1 y 3), dando como resultado una división entera con resultado `0`.

Hay que prestar especial atención a las combinaciones de tipos. El casting y la conversión implícita desempeñan un papel crucial para garantizar la corrección de los resultados.

2·3.3 Operadores enteros

Los enteros son el tipo de dato más importante en todos los lenguajes. Son la codificación más natural en los ordenadores. Muchas cosas que no son en sí mismas números acaban codificadas como enteros, para que podamos compararlos, sumarlos o restarlos. Las prioridades, por ejemplo, suelen ser simplemente `0`, `1`, `2`, ..., `10` o así.

Tenemos los operadores aritméticos básicos `+`, `-` (que también funciona como negación unaria), `*` (sí, la multiplicación es `*`, no `x`) y `/` (división entera):

```
int i = 0;
int j, k;

j = i + 1;
k = -j;  // unary minus: k is now -1
```

pero hay que tener cuidado aquí: siempre que todos los operandos son enteros, el resultado es también siempre un entero, así que

```
cout << 7 / 3 << endl; // will print 2, no fract. part
```

2·3.3.1 Módulo, o resto

Al dividir dos enteros, la parte fraccionaria se descarta (`7/3` da `2`). Para abordar esta limitación, existe un operador fundamental —argumentablemente incluso más ampliamente utilizado que las cuatro operaciones aritméticas básicas— el operador **módulo** o **resto** `%`, que devuelve el resto de la división entera entre dos enteros:

```
cout << 7 % 3 << endl; // will print 1
```

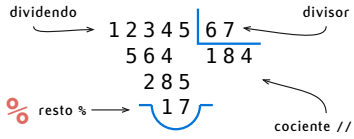
Esto produce el resto al dividir 7 entre 3. El operador módulo es particularmente útil porque nos permite responder preguntas como *¿es este número par?* (¿`n % 2` es igual a `0`?) o *¿acabamos de llegar a la columna 80, contando con pos?* (¿`pos % 80` es igual a `0`?).

La división y el módulo están ligados por una identidad que merece recordarse —es en sí misma una pequeña invariante, del tipo en que nos apoyaremos constantemente cuando lleguemos a los bucles—: para cualquier entero `n` y cualquier divisor no nulo `d`:

```
n == (n / d) * d + n % d
```

siempre se cumple. Lo que `/` descarta, `%` lo devuelve.

58: El operador módulo es de un valor incalculable: devuelve el resto de la división entera.
`12345 / 67` ⟶ `184`
mientras que
`12345 % 67` ⟶ `17`



Desde C++ 11, `/` trunca hacia cero y `%` sigue la misma regla, de modo que los operandos negativos se comportan de forma predecible:

```
cout << -7 / 2 << endl;  // -3
cout << -7 % 2 << endl;  // -1
cout <<  7 % -2 << endl; //  1
```

Antes de C++ 11, la dirección de redondeo para operandos negativos estaba definida por la implementación — una razón más para no confiar en código que dependa de ello sin verificar qué estándar está en vigor.

2·3.3.2 Incremento y decremento

Además del nuevo operador `%`, tenemos los operadores de pre/post-incremento y pre/post-decremento, `++i`, `i++`, `--i`, `i--`. Cada uno de los siguientes comienza de nuevo desde `i = 1`:

```
int i = 1;
// pre-increment:
cout << ++i << endl; // i becomes 2 first, then 2 is printed
```

```
int i = 1;
// post-increment:
cout << i++ << endl; // 1 is printed first, then i becomes 2
```

```
int i = 1;
// pre-decrement:
cout << --i << endl; // i becomes 0 first, then 0 is printed
```

```
int i = 1;
// post-decrement:
cout << i-- << endl; // 1 is printed first, then i becomes 0
```

¿Qué imprime el siguiente programa?

Nunca modifique la misma variable más de una vez dentro de una única expresión, como en `cout << i++ + i++;`. El estándar de C++ no garantiza un orden de evaluación aquí, por lo que el resultado puede diferir entre compiladores — o incluso entre niveles de optimización del mismo compilador.

C++
```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int i = 3;
7      cout << ++i << endl;
8
9      i = 3;
10     cout << i++ << endl;
11     cout << i << endl;
12
13     i = 3;
14     cout << i++ + i << endl;
15     cout << i << endl;
16
17     return 0;
18 }
```

Sin embargo, hay que ser consciente de la advertencia que el linter y el compilador emitirán en la línea 14:

Display
```
clang++: warning
warning - unsequenced modification and access to 'i' [-Wunsequenced]
```

Esta advertencia indica que el orden de las operaciones en la suma es indefinido entre diferentes compiladores y puede producir resultados distintos. Típicamente imprimirá 7 (4+3), ya que `i` se post-incrementa, es decir, se incrementa después de que se evalúa toda la adición.

2·3.3.3 Operadores de asignación compuesta

Y, aunque no son exclusivos de los enteros, los siguientes operadores también pueden usarse con números reales y `char`s (que se comportan como enteros pequeños en aritmética) —
`+=` `-=` `*=` `/=` `%=`, etc.
son atajos para una asignación combinada con una operación sobre la misma variable. Veamos un ejemplo, aplicando cada uno de ellos en orden:

`%=` — como `%` en sí mismo — exige operandos enteros: no se puede escribir `x %= 2.5;`. Para los números reales, el equivalente es la función `fmod` de la biblioteca `<cmath>`, que veremos de nuevo en la siguiente sección.

```
int i = 20;
i += 5; // i becomes 25
i -= 3; // i becomes 22
i *= 2; // i becomes 44
```

```
i /= 4; // i becomes 11
i %= 4; // i becomes 3
```

no es solo un atajo de escritura: el compilador también tiende a generar código más eficiente de esta manera.

Hay que tener precaución con la siguiente expresión:

```
result *= a + 2;
```

`result` se multiplica por la suma `a + 2`. Nótese que los operadores de asignación compuesta tienen menor precedencia que los operadores aritméticos, por lo que la sentencia anterior es equivalente a

```
result = result * (a + 2);
```

Y:

Combinado	Equivalencia
`y -= a / 2;`	`y = y - (a / 2);`
`a += b - 2;`	`a = a + (b - 2);`
`v *= 100 - p;`	`v = v * (100 - p);`
`a %= b - 1;`	`a = a % (b - 1);`
`a /= b + c;`	`a = a / (b + c);`

2·3.4 Asignación múltiple

C++ permite asignar un valor (o el resultado de una expresión) a múltiples variables simultáneamente. Es un atajo conveniente de uso frecuente que simplifica la escritura del código y es particularmente apropiado en las inicializaciones.

```
cnt = sum = i = 0;
```

asigna el valor del lado derecho a cada variable de su izquierda.

2·3.5 Desbordamiento de enteros

Cada tipo entero tiene un rango limitado (los tabulamos en breve), y los operadores aritméticos nunca comprueban si un resultado todavía cabe en él:

```
unsigned char u = 255;
u = u + 1;
```

En C, los caracteres comparten muchas características con los enteros. La suma anterior da la vuelta a `0`; esto constituye un comportamiento bien definido para los tipos sin signo, aunque es matemáticamente absurdo.

Y

```
int n = 2147483647; // INT_MAX
n = n + 1;
```

El desbordamiento de enteros con signo es comportamiento indefinido, que en la práctica normalmente resulta en `-2147483648` en máquinas modernas, es matemáticamente absurdo y produce un resultado confuso.

El desbordamiento cíclico para tipos sin signo está garantizado por el estándar; el desbordamiento de un tipo con signo no lo está — el compi-

lador tiene libertad para asumir que nunca ocurre, lo cual es precisamente lo que lo hace peligroso.

2·3.6 Complemento a dos

La razón de este comportamiento extraño en los enteros (no tan extraño en los enteros sin signo) se debe a su codificación. Un entero corto sin signo (omitimos el byte superior) como 3 se codifica simplemente como `00000011`, pero un entero como −3 se codifica como: `11111101`

Este es el **complemento a dos**: el bit más a la izquierda tiene un peso negativo — aquí, mirando un único byte, -128 en lugar de $+128$ — y los bits restantes se suman como de costumbre. Por tanto, `11111101` vale $-128 + 64 + 32 + 16 + 8 + 4 + 0 + 1 = -3$.

Binario	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1100	1101	1111
unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
int	0	1	2	3	4	5	6	7	−8	−7	−6	−5	−4	−3	−2	−1

Nadie suma ocho términos ponderados a mano cada vez, sin embargo. Hay un atajo mecánico que vale la pena memorizar para entender volcados de memoria o depurar código a nivel de bits. Para codificar $-n$:

1. escribir n en binario,
2. **invertir todos los bits** (los `0` se convierten en `1` y viceversa — este paso intermedio se llama el **complemento a uno**),
3. luego **sumar 1**:

```
  00000011   (3)                                          Display
  11111100   (flip every bit: one's complement)
+ 00000001
------------
  11111101   (-3, two's complement)
```

Esta pequeña receta también explica por qué el desbordamiento con signo parece tan sorprendente. Tomemos el mayor byte positivo, `01111111` (127 — exactamente el máximo de `char` en la tabla de abajo), y sumemos 1 de la manera ordinaria:

```
  01111111   (127)                                         Display
+ 00000001
------------
  10000000
```

Leído como un patrón **sin signo**, `10000000` es simplemente 128 — nada extraño ahí. Pero en el momento en que el bit más a la izquierda se activa, deja de ser un dígito ordinario y se convierte en el bit de signo, con valor -128: por lo que ese mismo patrón, leído como con signo, es -128 (el mínimo de `char`). Uno más que 127 es -128 — exactamente el **desbordamiento con signo** de la sección anterior, y exactamente por qué el estándar se niega a definirlo: los bits son inequívocos, pero su significado depende completamente de si el tipo es con o sin signo, y las dos lecturas no podrían divergir más. Lo mismo ocurre, un bit más ancho, cuando `int n = INT_MAX; n = n + 1;`: el bit de signo se activa y `n` se convierte en `INT_MIN`.

2·3.7 Punto de verificación

1) Describa la diferencia entre una palabra clave del lenguaje y un identificador definido por el programador.
2) ¿Cuál es la diferencia entre una línea de programa y una sentencia?
3) Explique por qué las variables se llaman «variables» y qué aspecto de ellas puede cambiar.
4) ¿Qué contienen las variables cuando declaramos su nombre y tipo?
5) ¿Cuáles de las siguientes son sentencias de asignación C++ legales?
   - 5.a) `a = 2;`
   - 5.b) `13 = x;`
   - 5.c) `2 = 2;`
   - 5.d) `x == 2;`
6) Liste todas las variables y literales que aparecen a continuación.

```
int main()
{
   int time;
   int space;

   time = 0;
   space = 10;
   cout << "Time is " << time << endl;
   cout << "Space is " << space << endl;

   return 0;
}
```

7) Escriba una sentencia de asignación múltiple que asigne `0` a las variables `counter`, `sum`, `totalPrice` y `totalTaxes`.
8) Escriba sentencias usando operadores de asignación combinados para realizar lo siguiente:
   - 8.a) Restar `-4` de `price`.
   - 8.b) Sumar `-6` a `x`.
   - 8.c) Multiplicar `totalTaxes` por $1.2 \cdot 10^{-3}$.
   - 8.d) Dividir `price` entre 1.5.
   - 8.e) Almacenar en `x` el resto de `x` dividido entre 7.
   - 8.f) Sumar 3 veces `totalTaxes` veces `price` a `total`.
   - 8.g) Sumar `y / 5` a `y`.
   - 8.h) Dividir `total` entre el resultado de `sume` menos `10`.
   - 8.i) Multiplicar `sum` por 6 veces `totalTaxes`.
9) ¿Qué mostrará el siguiente segmento de programa?

```
int a, b, c;
a = b = c = 7;
b *= 2;
```

```
c -= 4;
a += 4;
a /= 3;
cout << "a: " << a << endl << "b: " << b << endl << "c: " << c << endl;
```

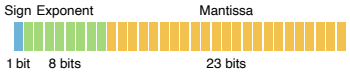
2·3.8 Operadores de punto flotante

Los valores de punto flotante que C++ puede manejar directamente se encuentran en estos rangos:

	float	double	long double
bytes	4	8	16
exp bits	8	11	15
mant bits	23	52	112
exp bias	127	1023	16383
min	-3.40282347e+38	-1.7976931348623157e+308	-1.189731495357231765085759326628007 02e+4932
MAX	3.40282347e+38	1.7976931348623157e+308	1.189731495357231765085759326628007 02e+4932

Para `double`, la representación interna de bits sería la siguiente:

- **`float`** 4 bytes:



- **`double`** 8 bytes:



- **`long double`** 16 bytes:



Como `long double` (128 bits), el número `1.0` se codifica internamente como:
1 bit de signo + 15 de exponente + 112 de fracción.
El sesgo del exponente es 16383 = `0x3fff`:

```
0 3fff 0000 0000 0000 0000 0000 0000 0000                    Display
= 1.0
```

La codificación y la representación interna de los valores de punto flotante es sustancialmente más intrincada que la de los enteros.

En la mayoría de los casos, utilizaremos la biblioteca `#include <cmath>` para operaciones de punto flotante. Sin embargo, algunas operaciones básicas, como elevar un float al cuadrado o al cubo, se ejecutan significativamente más rápido cuando se realizan directamente: `r * r` en lugar de `pow(r, 2)`, o `r * r * r` en lugar de `pow(r, 3)`. La biblioteca `cmath` proporciona definiciones para funciones como `sqrt(x)` (la raíz cuadrada de `x`) y `sin(x)`, pero no incluye ciertas funciones matemáticas trascendentes, como el factorial, que implementaremos nosotros mismos a lo largo de este curso.

2·3.9 Visualización formateada

Para que cada valor ocupe un número fijo y predecible de columnas al imprimirse, C++ proporciona dos manipuladores de flujo útiles:

`setw(w)` establece el ancho de la *siguiente* salida únicamente, rellenando con espacios (alineado a la derecha por defecto) hasta que ocupa `w` caracteres. A diferencia de la mayoría del estado del flujo, el ancho se restablece tras cada uso, por lo que `setw()` debe llamarse de nuevo antes de cada valor que se quiera rellenar.

`setprecision(w)` establece cuántos dígitos se muestran para los números de punto flotante. Por defecto, `w` es el número total de *dígitos significativos* mostrados.

`fixed` combinado con `setprecision(w)`, `w` se convierte en el número de dígitos *después* del punto decimal. `fixed` indica que la salida de punto flotante debe imprimirse en notación de punto fijo, o decimal.

`left` hay ocasiones en que se puede querer forzar que un valor se imprima en el lado izquierdo de su campo, rellenado con espacios a la derecha. Para ello se puede usar el manipulador `left`. Permanece en vigor hasta que se use el manipulador `right` para restablecerlo. `left`/`right` pueden usarse con cualquier tipo.

Estos manipuladores• están definidos en la cabecera `<iomanip>`.

• Los manipuladores de flujo son funciones ordinarias que, cuando se envían a un flujo mediante `<<`, cambian el estado de formato interno del flujo en lugar de imprimir un valor por sí mismos — aquí modifican el estado de `cout`.

Con los floats, por defecto, `cout` muestra solo 6 dígitos significativos, independientemente de cuántos almacene realmente el tipo. Si se lee un valor y se imprime de nuevo sin mayor consideración, puede que no se reconozca el propio número:

```cpp
// iomanip.cpp
// juanfc 2026-09-09
//
#include <iostream>
#include <iomanip>   // for setprecision
using namespace std;

int main()
{
    double x = 3.98765999;

    cout << "'" << setw(8) << 42 << "'" << endl;     // '      42'(spaces padding)
    cout << x << endl;                               // 3.98766   (general)
    cout << setprecision(2) << x << endl;            // 4      (2 digits rounding)
    cout << setprecision(3) << x << endl;            // 3.99   (3 digits in total)
    cout << fixed << setprecision(3) << x << endl;   // 3.988  (3 decimals)

    return 0;
}
```

Otro ejemplo:

```cpp
// readFloatPrecision.cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  float f;
  cout << "Enter a real number: ";
  cin >> f;

  cout << "Without setprecision: " << f << endl;                    // 6 significant digits
  cout << "With setprecision(9): " << setprecision(9) << f << endl; // up to float's own precision
  return 0;
}
```

Introduzca, por ejemplo, `3.14159265` (tantos dígitos de π como recuerde): la primera línea imprime un `3.14159` redondeado, mientras que la segunda revela lo que `f` almacena realmente — `3.14159274`, lo más cerca que un `float` puede llegar a π, y ni un dígito más, ya que sus 23 bits de mantisa solo compran unos 9 dígitos significativos fiables. `setprecision`, de `<iomanip>`, es **persistente**: una vez establecido, permanece en efecto para cada `cout` posterior hasta que se cambie de nuevo — así que restablézcalo después si el resto de la salida no debe quedar con 9 dígitos.

Notación científica la mantisa se normaliza a exactamente un dígito distinto de cero antes del punto decimal, de modo que $1 \leq |m| < 10$, y el exponente puede ser cualquier entero. Ejemplo: `47000` $\longrightarrow 4.7 \times 10^4$.

Notación de ingeniería el exponente está restringido a múltiplos de 3, por lo que la mantisa se extiende hasta $1 \leq |m| < 1000$. Ejemplo: `47000` $\longrightarrow 47 \times 10^3$. El propósito de los pasos de `×3` es que los exponentes se alinean con los prefijos SI (10^3 = kilo, 10^6 = mega, 10^{-3} = mili, 10^{-6} = micro…), por lo que 47×10^3 se lee directamente como 47 k.

La notación de ingeniería es, por tanto, una variante de la notación científica en la que el exponente es siempre un múltiplo de tres (y la mantisa se encuentra en el rango $1 \leq |m| < 1000$), de modo que cada potencia de diez corresponde a un prefijo SI (Sistema Internacional) como kilo, mega, mili o micro. En C++, tenemos `std::scientific`

2·3.9.1 Punto de verificación

1) Escriba sentencias `cout` con manipuladores de flujo que realicen lo siguiente:

   1.a) Mostrar el número `34.789` en nueve espacios con dos decimales de precisión.

   1.b) Mostrar el número `7.0` en un campo de ocho espacios con tres dos decimales de precisión. Deben mostrarse el punto decimal y los ceros finales.

   1.c) Mostrar el número `3.289e+12` en notación de punto fijo.

   1.d) Mostrar el número `76` justificado a la izquierda en un campo de cinco espacios.

2) El siguiente segmento de programa convierte un ángulo en grados a radianes.

```
const float PI = 3.14159;
float degrees, radians;

cout << "Enter an angle in degrees and I will convert it" << endl;
cout << "to radians for you: ";
cin >> degrees;
radians = degrees * PI / 180;

// Display the value in radians left-justified, in fixed-point
// notation, with four decimal places of precision,
// in seven spaces wide.
```

2·3.9.2 Límites para los contenidos de los tipos básicos

Puede encontrar los límites de rango de los diferentes tamaños de enteros y floats con este programa:

```C++
#include <iostream>
#include <climits>
#include <cfloat>
using namespace std;

int main()
{
    cout << USHRT_MAX << endl;  //  65 535
    cout << SHRT_MIN << endl;   // -32 768
    cout << SHRT_MIN << endl;   //  32 767
    cout << INT_MIN << endl;    // -2 147 483 648
    cout << INT_MAX << endl;    //  2 147 483 647
    cout << ULONG_MAX << endl;  //  18 446 744 073 709 551 615
    cout << FLT_MAX << endl;    //  3.40282e+38
    cout << DBL_MAX << endl;    //  1.79769e+308
    cout << LDBL_MAX << endl;   //  1.18973e+4932
    return 0;
}
```

Una información no tan frecuentemente necesaria es la cantidad de bytes que demanda cada variante:

```C++
// sizesof.cpp
#include <iostream>
using namespace std;
int main()
{
    cout << "Sizes in bytes of:" << endl;
    cout << "char:         " << sizeof(char)        << endl;  //  1
    cout << "short int:    " << sizeof(short)       << endl;  //  2
    cout << "int:          " << sizeof(int)         << endl;  //  4
    cout << "long int:     " << sizeof(long)        << endl;  //  8
    cout << "float:        " << sizeof(float)       << endl;  //  4
    cout << "double:       " << sizeof(double)      << endl;  //  8
    cout << "long double:  " << sizeof(long double) << endl;  // 16
    return 0;
}
```

Y en general, los tamaños y capacidades para todos los tipos básicos son los siguientes:

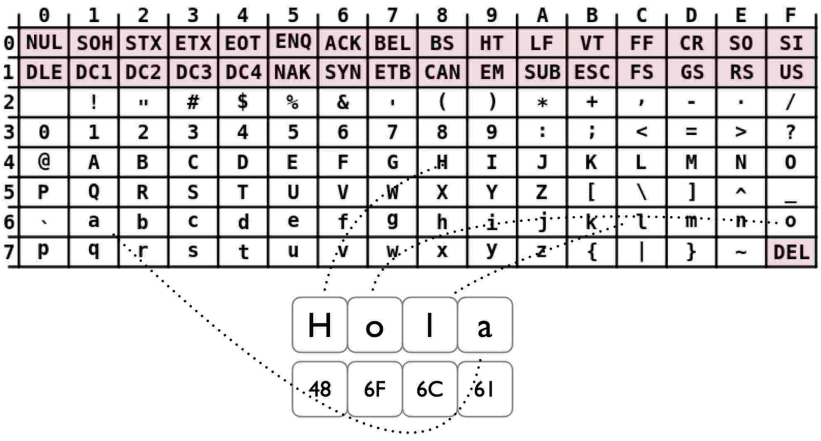
	bytes	mín	máx
bool	1	false	true
char	1	-128	127
unsigned char	1	0	255
short	2	-32,768	32,767
unsigned short	2	0	65,535
int	4	$-2{,}147_{1}483{,}648$	$2{,}147_{1}483{,}647$
unsigned	4	0	$4{,}294_{1}967{,}295$
long	4	$-2{,}147_{1}483{,}648$	$2{,}147_{1}483{,}647$
long long	8	$-9_{3}223{,}372_{2}036{,}854_{1}775{,}808$	$9_{3}223{,}372_{2}036{,}854_{1}775{,}807$
unsigned long long	8	0	$18_{3}446{,}744_{2}073{,}709_{1}551{,}615$
float	4	-3.40282e+38	3.40282e+38
double	8	-1.79769e+308	1.79769e+308
long double	16	-1.18973e+4932	1.18973e+4932

2·4 Caracteres y cadenas

Hemos aprendido que hay dos tipos de datos básicos: los números y los caracteres. Internamente, por supuesto, los caracteres se codifican como hemos visto en Sección 2·2.1 como enteros simples, pero su tratamiento no incluye la suma o la división, sino operaciones para leer/escribir caracteres desde/hacia el teclado/pantalla. Para referirnos a un único carácter, encerramos su literal entre comillas simples:

```
char letter        = 'A',
     anotherLetter = '-';
```

Los caracteres tienen una relación fundamental con los enteros porque se representan internamente como valores enteros. Cada carácter imprimible, junto con muchos caracteres no imprimibles, tiene asignado un código numérico único. El esquema de codificación de caracteres más ampliamente utilizado es ASCII, que significa American Standard Code for Information Interchange (Código Estándar Americano para el Intercambio de Información). Cuando se almacena un carácter en memoria, se almacena el código numérico correspondiente en lugar del carácter en sí. Cuando se le indica al ordenador que imprima un valor de carácter, recupera el código entero y lo mapea a la entrada correspondiente en la tabla ASCII, luego muestra el carácter en pantalla. El Apéndice A, al final de este texto, contiene el conjunto completo de caracteres ASCII, lo que permite identificar qué valor entero representa cada carácter. Nótese que el entero 65 representa la A mayúscula, el 66 representa la B mayúscula, y así sucesivamente.




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

int main()
{
   char aLetter;

   // We will not do this in our programs, but this is possible in C
   aLetter = 65;              // 65 is the ASCII code for the character A
   cout << aLetter << endl;  // A
   cout << char(67) << endl; // (integer forced to be a char) prints C

   return 0;
}
```

Un enfoque sencillo y eficaz para convertir una letra minúscula o mayúscula a su correspondiente mayúscula o minúscula es sumar la distancia del código ASCII entre ellas, ya que están dispuestas en paralelo:

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

int main()
{
    char c;
    cout << "Enter a letter: ";
    cin >> c;

    char uppercaseC = c;
    if ('a' <= c and c <= 'z') {
       uppercaseC += ('A' - 'a');
     }

    cout << "Uppercase of " << c
         << " is " << uppercaseC << endl;

    return 0;
}
```

Esto es posible gracias a las conversiones implícitas (`char` ⟶ `int` para las operaciones de suma y resta, con el `int` resultante convertido de nuevo a `char` para la asignación)◦.

◦ Algunos prefieren usar la distancia (32) como un número mágico que memorizan para realizar esta conversión mayúsculas/minúsculas. Evitar depender de constantes memorizadas oscuras hace que los programas sean más legibles y se autodocumenten.

2·4.1 Punto de verificación

1) ¿Cuáles de los siguientes son nombres de variables C++ ilegales, y por qué?

   1.a) `n`

   1.b) `1length`

   1.c) `sept26`

   1.d) `theNameOfThisVariableIsVerbose`

   1.e) `r&d`

   1.f) `_gradeIP`

   1.g) `_0_1_2_`

2) ¿Se consideran iguales las variables `Height` y `height`?

3) ¿Por qué preferimos los enteros con signo (`int`) a los enteros sin signo (`unsigned`) cuando representamos valores sin signo?

4) ¿Cómo combinaría la siguiente declaración e inicialización de variable en una sola sentencia?

```cpp
int counter;
counter = 20;
```

5) ¿Cómo combinaría las siguientes definiciones de variables en una sola sentencia?

```cpp
float xForce = 0;
float yForce = -10;
float zForce = 6;
```

6) ¿Cuáles son los códigos ASCII de los siguientes caracteres?

   6.a) `X`

6.b) `)`

6.c) `-`

6.d) `0`

7) ¿Qué tiene de incorrecto la siguiente sentencia de programa?: `char c = "X";`

8) ¿Cuántos bytes usa cada uno de los siguientes literales?

8.a) `'A'`

8.b) `"a"`

8.c) `"length"`

8.d) `'\n'`

9) ¿Qué archivo de cabecera hay que incluir para usar objetos string? ¿Es necesario en todos los compiladores?

10) ¿Qué valor se asignará a `x` en cada una de las siguientes sentencias?

10.a) `x = 7 + 3;`

10.b) `x = 7 / 3;`

10.c) `x = 7 % 3;`

11) ¿Es el siguiente un ejemplo de división entera o de punto flotante? ¿Qué valor se mostrará?: `cout << 16 / 3 << endl;`

2·5 Enumeración de los valores que puede contener un tipo

Imagine que queremos trabajar con unos pocos colores. ¿Cómo declararía los posibles contenidos de una variable de tipo `colour` para que cuando se asigne un color como rojo o verde, el código sea fácil de seguir? Si declaramos

```
int colour;

colour = 1;
colour = 3;
```

esto no es fácil de entender.

Cuando los valores que manejamos no son ni números ni letras y son muy pocos, podemos enumerar los valores posibles y referirnos a cada uno de ellos usando etiquetas o identificadores que, a partir de ese punto en nuestro programa, representarán un valor concreto adecuado para asignar a ese tipo enumerado.

```
enum TColour {BLUE, GREEN, RED, BROWN};
```

Aquí tiene 5 nuevos nombres, identificadores, para su programa: las 4 nuevas etiquetas para cada uno de los valores y el nombre del nuevo tipo.

Ejemplos típicos en los que se puede ver lo útiles que pueden ser los tipos enumerados:

```
enum TMonth     {JANUARY=1, FEBRUARY, MARCH, APRIL,
                 MAY, JUNE, JULY, AUGUST,
                 SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER};

enum TWeekDay   {MONDAY=1, TUESDAY, WEDNESDAY,
                 THURSDAY, FRIDAY, SATURDAY, SUNDAY};

enum TPlanets   {MERCURY, VENUS, EARTH, MARS,
                 JUPITER, SATURN, URANUS, NEPTUNE};

enum TState     {LOW=-10, HIGH=10, UNDEF};
```

y un ejemplo de su uso:

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

enum TColour {BLUE, GREEN, RED, BROWN};

int main()
{
    TColour colorSky, colorGround;
    colorSky = BLUE;
    colorGround = BROWN;

    cout << colorSky << endl;
    cout << colorGround << endl;

    return 0;
}
```

Si no asignamos valores a las etiquetas, los enums se codifican internamente como la secuencia 0, 1, 2, ... Sin embargo, podemos asignar cualquier valor entero a alguna o todas las etiquetas. Una etiqueta sin valor explícito obtiene el valor anterior más uno (por ejemplo, en el enum `TState {LOW=-10, HIGH=10, UNDEF}`, `UNDEF` es 11). Esto es útil cuando esos valores enteros tienen significado semántico.

2·5.1 Punto de verificación

1) Identifique todos los errores en la siguiente declaración.

```
Enum Pet = { "dog", "cat", "bird", "fish" }
```

2) Defina un tipo enumerado para representar las calificaciones educativas españolas, desde `SUSPENSO` hasta `SOBRESALIENTE`, asignando a cada etiqueta el valor máximo que representa para que más adelante en un programa pueda comparar puntuaciones con ellas para tomar decisiones (por ejemplo, `mark > SUSPENSO`).

2·6 typedef: Definición de nuevos nombres para tipos

Las siguientes sentencias definen nuevos tipos basados en tipos previamente conocidos, o alternativamente proporcionan nuevos nombres para tipos existentes:

`typedef`

```
typedef int   TCounter;
typedef float TVol;
typedef TVol  TSpace;
```

A partir de este punto, `TCounter` es equivalente a `int`, pero declarar una variable usando `TCounter` mejora significativamente la legibilidad del programa:

```
TCounter cnt;
```

en comparación con declararla como `int`.

La primera razón para usar `typedef` es mejorar la legibilidad mientras simultáneamente se habilita la parametrización y se crean programas más flexibles. Por ejemplo, suponga que declara una serie de variables de tipo `TSurface` y luego descubre que es mejor manejar estas cantidades (y solo estas) como `double` en lugar de `float`. Solo necesita cambiar el tipo una vez en la

```
typedef double TSurface;
```

declaración.

```
float l, s, v;  // are poor declarations of...
typedef float TLength;
typedef float TSurface;
typedef float TVolume;
// latter you will build variables based on these new names of types
TLength l;
TSurface s;
TVolume v;
```

En un programa para manejar cantidades geométricas:

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

typedef int TBase;
typedef TBase TArea;
typedef TBase TVolume;
typedef TBase TLength;

int main()
{
    TLength a, b, c;
    TArea s1, s2;
    TVolume v;
    // …
    return 0;
}
```

O una situación en la que tiene un nombre de tipo engorroso como `unsigned short int`:

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

// typedefs
typedef unsigned short int Tushort;

int main()
{
    Tushort width = 5;
    Tushort length;

    length = 10;
    Tushort area  = width * length;

    cout << "width:  " << width  << endl;
    cout << "length: " << length << endl;
    cout << "area:   " << area   << endl;
}
```

Sin embargo, la importancia de `typedef` va más allá de estas dos consideraciones. Veremos más adelante que con los tipos complejos, el uso de alias (que es todo lo que `typedef` crea) es esencial.

2·7 Números aleatorios

Aunque los ordenadores no pueden generar números verdaderamente aleatorios, es importante tanto para los juegos como para las pruebas exponer nuestros programas a valores aleatorios. Los números aleatorios, o *números pseudoaleatorios* como se llaman correctamente, son valores de una secuencia matemáticamente generada que no siguen ningún patrón discernible y se garantiza de forma aproximada que son estocásticamente uniformes; es decir, sus valores están distribuidos de forma uniforme. Si ejecutamos nuestro programa y pedimos un valor aleatorio, siempre producirá el mismo resultado, y la segunda llamada producirá el mismo valor, y así sucesivamente. Para obtener valores diferentes cada vez que ejecutamos nuestro programa, usamos lo que se llama una *semilla*. Sembramos la secuencia proporcionando un punto de partida, y para garantizar que la secuencia difiere con cada ejecución, típicamente usamos la hora del sistema.

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    srand(time(0)); // seeds the seq. to the system time
    for (int i = 0; i < 10; ++i) {
        cout << rand() << endl;
    }
    return 0;
}
```

`rand()` devuelve un entero sin signo entre `0` y un entero máximo (`RAND_MAX`). Para generar un valor aleatorio dentro de un rango específico —por ejemplo, para simular el lanzamiento de un dado— usamos la fórmula:

```
number = (rand() % (maxVal - minVal + 1)) + minVal;
```

y para el lanzamiento de un dado específicamente:

```
cout << rand() % 6 + 1 << endl;
```

2·8 Precedencia de operadores

Cuando una expresión combina varios operadores, el compilador debe decidir cuál aplicar primero. Exactamente como en la aritmética ordinaria `2 + 5 * 3` significa `2 + 15` y no `21`, cada operador tiene una **precedencia**: cuanto mayor es su precedencia, más fuertemente se une a sus operandos y antes se aplica. Cuando dos operadores comparten la misma precedencia, una segunda regla, la **asociatividad**, desempata — para la mayoría de los operadores, de izquierda a derecha.

Precedencia	Operadores	Asoc.
mayor	`( )` agrupación, llamada a función, se evalúa antes que cualquier otro	
	`not` unary `-` `++` `--`	derecha
	`*` `/` `%`	izquierda
	`+` `-`	izquierda
	`<` `<=` `>` `>=`	izquierda
	`==` `!=`	izquierda
	`and`	izquierda
	`or`	izquierda
menor	`=` `+=` `-=` `*=` `/=` `%=`	derecha

Dos consecuencias de esta tabla merecen tenerse en cuenta. En primer lugar, **los operadores multiplicativos se unen antes que los aditivos**, por lo que `result = 2 + 5 * b;` se lee como `result = 2 + (5 * b);`: el producto se forma primero y solo entonces se suma a 2. En segundo lugar, **la asignación se encuentra en el nivel más bajo**, por lo que todo el lado derecho se evalúa antes de que se almacene nada — lo que también explica por qué una asignación compuesta como `x *= a + 2;` significa `x = x * (a + 2);`, no `x = (x * a) + 2;`.

Asociatividad. La mayoría de los operadores binarios son **asociativos por la izquierda**, por lo que una cadena se agrupa desde la izquierda:

```
10 - 4 - 3   // means (10 - 4) - 3  ->  3,  not  10 - (4 - 3)  ->  9
```

La asignación es la excepción notable: es **asociativa por la derecha**, lo que es precisamente lo que hace que una asignación múltiple funcione. `a = b = c = 7;` se agrupa como `a = (b = (c = 7));`, por lo que el 7 viaja de derecha a izquierda a las tres variables.

Relacionales antes que lógicos. Las comparaciones se unen más fuerte que `and`/`or`, por lo que una prueba de rango se lee de forma natural sin paréntesis en absoluto:

```
if (x >= 0 and x <= 100) ...   // (x >= 0) and (x <= 100)
```

El `not` unario es la trampa aquí: al ser unario, se une **muy** fuertemente — más que `==` — por lo que captura solo el token inmediatamente a su derecha:

```
not x == y     // means (not x) == y  — almost certainly NOT what you meant
not (x == y)   // this is the intended test
```

Siempre que el agrupamiento no sea obvio a primera vista, no obligue al lector (ni a usted mismo) a recordar toda la tabla — añada paréntesis. No tienen coste en tiempo de ejecución y expresan la intención claramente◆.

◆ Una trampa de precedencia merece una mención especial, porque involucra un operador que ya se usa constantemente: `<<`. Se une más fuertemente que los operadores de comparación, por lo que `cout << x == 3;` se lee como `(cout << x) == 3;` y no hará lo que se espera. Escriba `cout << (x == 3);` cuando quiera imprimir el resultado de una comparación.

2·8.1 Punto de verificación

1) En la expresión `2 + 5 * b`, ¿qué operación se realiza primero y por qué?

2) Reescriba cada expresión con paréntesis explícitos que muestren el agrupamiento que realmente usa el compilador:

   2.a) `a - b - c`

   2.b) `a = b = c`

   2.c) `x < y and y < z`

   2.d) `not a == b`

3) ¿Qué valor produce `10 - 4 - 3`? ¿Qué produciría si `-` fuera asociativo por la derecha?

4) Explique por qué `a = b = c = 7;` almacena 7 en las tres variables. ¿Qué propiedad lo hace posible?

5) Añada los paréntesis mínimos necesarios para que esta prueba signifique «`x` está fuera del rango `0` a `100`»:

```
not x >= 0 and x <= 100
```

6) ¿Qué hace realmente `cout << x == 3;` y cómo lo corregiría para que imprima si `x` es igual a 3?

7) Mostrando el orden de las operaciones paso a paso, ¿qué valor se almacena en `r`?

```
int a = 2, b = 3, c = 4;
int r = a + b * c - 6 / 3;
```

8) ¿Cuáles son los valores de las siguientes expresiones?

   8.a) `9 - 1 * 5`

   8.b) `6 / 2 - 4`

   8.c) `9 + 9 * 2 - 6`

   8.d) `5 + 19 % 3 - 1`

   8.e) `8%(6+2)*22`

   8.f) `44/(11-4)`

   8.g) `(8+17) % 1-3`

   8.h) `(3-9)*(9+9)/3`

2·9 Entrada y Salida Básicas

Un programa que no puede comunicarse con su entorno externo tiene una utilidad práctica limitada. Presentamos los flujos estándar `cin` y `cout`, que permiten a los programas leer la entrada del teclado y escribir la salida en la pantalla. Aunque existen numerosos métodos alternativos para interactuar con el ordenador, nos centramos en el enfoque más universalmente aplicable: la comunicación terminal basada en texto. Esta elección refleja tanto la simplicidad como la amplia aplicabilidad en dominios de aplicación diversos y en evolución. Abordar los diálogos gráficos y las interfaces de usuario complejas distraería de los principios fundamentales de la programación. De hecho, el mecanismo de interacción mencionado (`cin`/`cout`) se usará en los programas iniciales. En los ejercicios posteriores

de este curso, típicamente asignaremos valores a las variables relevantes directamente dentro del programa usando «inicialización rápida», lo que nos permite verificar la funcionalidad de nuestro algoritmo de forma más eficiente en casos importantes sin la repetitiva sobrecarga de la entrada manual de datos cada vez que ejecutamos el programa.

59: El teclado y la pantalla son la entrada/salida estándar en nuestro lenguaje



C++ proporciona, de serie, el código necesario para leer y escribir los tipos básicos desde y hacia la «entrada/salida estándar»:

- `char`
- `int` (y sus diferentes variantes)
- `float` (y sus diferentes variantes)

```
int i;
char c;
float f;

cin >> i; // will halt the program and wait for you to enter a valid int
cin >> c; // will halt the program and wait for you to enter a valid char
          // however, it will never read whitespace characters:
          //      not ' ', not newlines, not tabs
cin >> f; // will halt the program and wait for you to enter a valid real
```

Pruébelo con:

```
// basicReading.cpp                                                    C++
#include <iostream>
using namespace std;
int main()
{
  int i;
  char c;
  float f;

  cin >> i; cout << "i: " << i << endl;
  cin >> c; cout << "c: " << c << endl;
  cin >> f; cout << "f: " << f << endl;
  return 0;
}
```

y no dude en intentar engañarlo introduciendo el tipo incorrecto de datos, para ver qué hace su programa en cada caso.

Recuerde que `cin` no lee los caracteres de espacio en blanco, que incluyen el carácter espacio (representado como `' '`), el carácter de nueva línea y el carácter de tabulación. Si proporciona solo espacios o presiona enter repetidamente, `cin` seguirá esperando la entrada. Lo mismo se aplica a la tecla de tabulación (la tecla horizontal larga en el lado izquierdo de su teclado).

2·9.1 Lectura/escritura de caracteres y cadenas

Recuerde que `cin` omite e ignora cualquier carácter de espacio en blanco inicial (espacios, tabulaciones o saltos de línea) al leer datos. Sin embargo, una vez que encuentra el primer carácter no en blanco y comienza a leer, se detiene cuando llega al siguiente carácter de espacio en blanco.

Para manejar secuencias de caracteres, usamos objetos `string`. Son mucho más convenientes que el enfoque clásico de usar arrays de caracteres◇. Las implementaciones más populares de C++ (GCC, Clang y MSVC) no requieren `#include <string>` al principio, ya que `string` se incluye transitivamente a través de `<iostream>`, que siempre incluimos. Por lo tanto, a menos que el compilador genere un error al declarar una cadena,

◇ clásicamente, C usa arrays de caracteres para manejar cadenas, pero los arrays son menos prácticos y generan una complejidad significativa cuando se quieren manipular de diversas formas

se puede omitir. Solo necesitamos saber que las cadenas pueden leerse y escribirse usando `cin` y `cout`, y que los literales de cadena se encierran entre comillas dobles:

```
string myName = "Juan",
       yourName = ""; // unnecessary, as strings are
                      // always guaranteed to initialise empty
```

Si necesitamos leer una cadena que incluya espacios, como `"Juan Antonio"` o similares, necesitaremos usar un enfoque diferente:

```
string yourAddress;
getline(cin, yourAddress); // will include everything the user inputs
                           // until they press enter
```

Cuando C++ encuentra algo que no puede analizar, no le dice educadamente que ha cometido un error y le pide que lo intente de nuevo: extrae silenciosamente todo lo que puede y continúa — de forma silenciosa, y habitualmente incorrecta, a menos que lo comprobemos. Lo que «todo lo que puede» significa depende del error:

- Una letra extraña en medio de un número (`12a34` tecleado para un `int`) detiene la lectura en la letra: `i` se convierte en `12`, y `a34` queda esperando en el búfer de entrada para el **siguiente** `cin`, que se atascará con él de inmediato.
- Teclear un valor con punto decimal donde se espera un `int` (`3.14`) se comporta de la misma manera: `i` se convierte en `3`, y `.14` queda atrás para arruinar la siguiente lectura.
- Un valor demasiado grande para el tipo destino (mayor que `INT_MAX`, por ejemplo) es diferente: `cin` establece su **estado de fallo** y fija la variable al valor extremo del tipo (`INT_MAX` o `INT_MIN`). Una vez en estado de fallo, `cin` rechaza **toda extracción posterior** — el programa parecerá que omite silenciosamente el resto de su entrada — hasta que se llame a `cin.clear()`.

Este último caso representa una de las fuentes más comunes de errores «mi programa simplemente deja de funcionar» entre los principiantes: compruebe siempre `if (not cin)` inmediatamente después de leer cualquier entrada del usuario, antes de confiar en el valor que acaba de leer.

■ Exploraremos cómo comprobar (`if`) el estado o los valores de los objetos en medio del programa muy pronto.

2·9.2 Punto de verificación

1) ¿Cómo se representaría el siguiente número en notación científica en notación de ingeniería con un factor de escala de potencia de diez? `6.31 × 1017`

2) ¿Qué mostrará el siguiente código?

```
int number;

number = 3.987:
cout << number << endl;
```

3) ¿Por qué existe la notación de ingeniería?

4) ¿Cuál es la diferencia entre `cin >> s;` y `getline(cin, s);`?

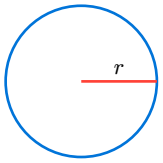
Tarea 1: Fundamentos. Programas secuenciales: aritmética y E/S estándar

Escribe los siguientes programas con los nombres indicados. Esta tarea cubre constantes, expresiones aritméticas y entrada/salida estándar.

1 `01.constants.cpp` Define las siguientes constantes en C++:

1. `MAXNUMBEROFCHARS` con el valor 256,
2. `PI` como `3.1416`,
3. `ENDOFLINE`, un `char` con el valor `'\n'`,
4. una constante que represente la distancia entre letras mayúsculas y minúsculas en la tabla ASCII,

2 `02.circle.cpp` El siguiente programa pide al usuario el radio de un círculo e imprime su área:



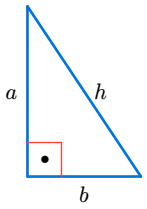

```cpp
#include <iostream>
using namespace std;

const float PI = 3.141592;

int main()
{
    float radius, area;
    cout << "Enter the circle radius: ";
    cin >> radius;
    area = PI * radius * radius;
    cout << "The area of a circle with radius " << radius
         << " is " << area << endl;
    return 0;
}
```

Modifícalo para que en su lugar pida el radio y la altura de un cilindro e imprima su volumen ($V = S_b \times h$, donde S_b es el área de la base).

3 `03.rightTriangle.cpp` Escribe un programa que pida al usuario los dos catetos a y b de un triángulo rectángulo, y calcule e imprima la hipotenusa h. Incluye `<cmath>` para usar `sqrt()`. La fórmula es:



$$h = \sqrt{a \cdot a + b \cdot b}$$

HINT: en general es preferible calcular `x*x` en lugar de llamar a `math::pow(x, 2)`, tanto por simplicidad como por eficiencia.

4 `04.celsius.cpp` Escribe un programa que pida al usuario una temperatura en grados Celsius (C) e imprima el equivalente en Fahrenheit (F). La fórmula de conversión es $F = \frac{9}{5} \cdot C + 32$. › Sol

5 `05.prices.cpp` Escribe un programa que pida al usuario el precio de un producto y un porcentaje de descuento, e imprima el precio final tras aplicar el descuento. › Sol

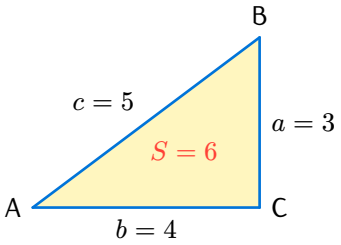
```
Price: 200
Discount (%): 15
Final price: 170
```

6 `06.seconds.cpp` Escribe un programa que pida al usuario un número de segundos transcurridos desde medianoche e imprima el tiempo equivalente en horas, minutos y segundos. › Sol

```
Seconds: 3661   ->  1h 1m 1s                                    example
Seconds: 7322   ->  2h 2m 2s
Seconds: 60     ->  0h 1m 0s
```

7 `07.heron.cpp` Escribe un programa que pida al usuario los tres lados de un triángulo e imprima su perímetro y su área usando la fórmula de Herón:

$$S = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = \frac{a+b+c}{2}$$



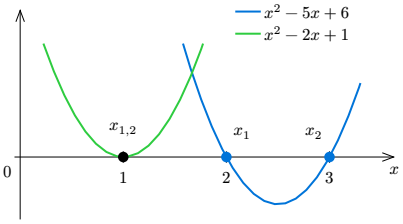
Incluye `<cmath>` para usar `sqrt()`. › Sol

```
Side a: 3  Side b: 4  Side c: 5                                 example
Perimeter: 12  Surface: 6
```

8 `08.digits.cpp` Escribe un programa que pida al usuario un número entero de cuatro cifras e imprima sus dígitos separados por espacios. Usa únicamente división entera y el operador módulo — sin cadenas ni arrays. › Sol

```
Enter a 4-digit number: 1234  ->  1 2 3 4                       example
Enter a 4-digit number: 4001  ->  4 0 0 1
```

9 `09.quadeq.cpp` Escribe un programa que pida al usuario los coeficientes a, b y c de una ecuación de segundo grado $ax^2 + bx + c = 0$ e imprima sus dos raíces reales usando la fórmula:



$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Puedes asumir que el discriminante es siempre no negativo. › Sol

```
a: 1  b: -5  c: 6   ->  x1: 3  x2: 2                            examples
a: 1  b: -2  c: 1   ->  x1: 1  x2: 1
```

10 `10.remainder.cpp` Escribe un programa que pida al usuario dos números enteros e imprima su cociente y el resto de la división entera, así como el resultado exacto como número real. › Sol

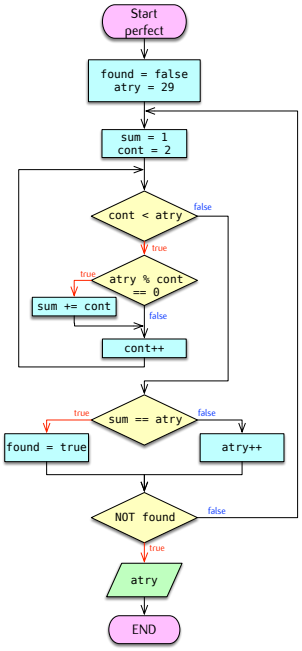
```
Dividend: 17  Divisor: 5 ->                                     example
Quotient: 3  Remainder: 2  Exact: 3.4
```

o O o

2·10 Flujo de control

Por defecto, las instrucciones se ejecutan secuencialmente de arriba a abajo. Las estructuras de control de flujo —concretamente la selección y la iteración— permiten que los programas sigan caminos de ejecución alternativos y repitan operaciones, lo que es esencial para expresar lógica algorítmica y crear programas que respondan de forma inteligente a diferentes condiciones.

64: Diagrama de flujo de control de un algoritmo para determinar si un número es perfecto



¿Por qué solo estos dos ingredientes adicionales —selección e iteración— sobre la secuenciación simple? El **teorema de Böhm–Jacopini** (Böhm, Jacopini 1966) responde a esto con precisión: cualquier función computable puede expresarse usando únicamente **secuencia**, **selección** e **iteración**, sin necesidad de saltos no estructurados (`goto`). Lo que un programa puede calcular saltando de un punto a otro, también puede calcularlo usando solo estas tres construcciones, a lo sumo a costa de algunas variables de control adicionales o de duplicar parte del código.

Esta es precisamente la razón por la que este curso prohíbe `goto`, `break` y `continue`: son todas formas controladas de salto, y el teorema garantiza que prescindir de ellas no nos cuesta nada en potencia expresiva. Piénsalo como si te dijeran que puedes construir cualquier forma usando solo tres tipos de piezas de Lego —individualmente poco llamativas, pero cuya **composición** (una tras otra, anidadas unas dentro de otras, repetidas) es suficiente para alcanzar cualquier cosa que un montón caótico de piezas de formas extrañas y atajos podría conseguir.

2·11 Expresiones booleanas

Toda decisión que toma un programa depende de una condición: una expresión que se evalúa como `true` o `false`. Examinamos cómo construir expresiones booleanas combinando comparaciones relacionales con los operadores lógicos `and`, `or` y `not`. Por ejemplo, `if (x > 10) {...`. La comparación `x > 10` se evalúa como `true` si y solo si el valor almacenado en `x` (que debe ser de un tipo compatible con comparaciones relacionales con enteros) es mayor que el entero `10`. En caso contrario ($x \leq 10$), la expresión booleana (`x > 10`) se evalúa como `false`.

```
x > 5
```

Estamos acostumbrados a expresiones aritméticas como:

$$2 + 4 \to 6$$

En programación, también nos familiarizamos con expresiones de comparación como:

$$6 > 5 \to \texttt{true}$$

En las comparaciones, el resultado de estas expresiones es un valor booleano (siempre `true` o `false`).

En C++ disponemos de los siguientes operadores de comparación:

```
a  > b
a <= b  // is a less than or equal to b
a == b  // is equal to b (be careful with double ==)
a != b  // is a different than b
// etc
```

Los valores booleanos no se producen únicamente mediante comparaciones. En general, todo lo siguiente produce resultados booleanos (`true` o `false`):

- Una constante booleana (`true` o `false`)
- Una variable booleana
- Una expresión con operadores relacionales (`>`, `<`, etc.)
- Operaciones lógicas (`and`, `or`) entre expresiones booleanas. En matemáticas, una **disyunción** representa la relación «o» y se denota con el símbolo $\vee$, mientras que una **conjunción** (donde ambos lados deben cumplirse simultáneamente) representa el «y» y se denota con el símbolo $\wedge$.
- La negación de una expresión booleana (en matemáticas, `not` se representa como $\neg$)

Por ejemplo:

```
bool isEven = (a % 2 == 0);
bool aprobado = score >= 5;
bool notable = score >= 7 and score <= 8; // < == >.. always 2 operands
bool isSomething = (a == 2 or c != 10) and (not b > 5);
```

Nótese que los operadores relacionales encadenados no están permitidos en este lenguaje y deben escribirse por separado:

~~`7 <= score <= 8;`~~ es `7 <= score and score <= 8;`

2·11.1 Visualización de booleanos

C++ puede leer un `bool` del mismo modo que puede imprimirlo. Por defecto, lee e imprime `0`/`1`; si insertamos `boolalpha` en el flujo previamente, lee e imprime `false`/`true`. Prueba:

```
// 01.boolReadingPrinting.cpp
// juanfc 2026-09-09
//

#include <iostream>
using namespace std;

int main()
{
  bool b = true;
  bool c = false;
  cout << b << endl;                  // prints: 1
  cout << c << endl;                  // prints: 0

  cout << boolalpha;                  // sticky: affects all later output
  cout << b << endl;                  // prints: true
  cout << c << endl;                  // prints: false

  cout << noboolalpha << b << endl;   // prints: 1 (back to the default)

  cout << "Enter true or false: ";
  cin >> boolalpha >> b;              // cin has its own flag: reads true/false
  cout << boolalpha << b << endl;     // prints: true or false

  return 0;
}
```

2·11.2 Punto de control

Supongamos que tenemos:

```
int  n;
char c;
```

Construye una expresión que sea `true` cuando:

1) `n` es mayor que 5 (Respuesta: `n > 5`)
2) `n` tiene dos dígitos
3) `c` es una letra mayúscula
4) `c` es una letra
5) `c` es una vocal
6) `n` es impar
7) `n` termina en `0`
8) `n` termina en `00`

2·11.3 Leyes de De Morgan

Entre las leyes que permiten manipular correctamente las expresiones booleanas, las leyes de De Morgan (De Morgan 1847) son especialmente valiosas para los programadores: establecen que **la negación de una conjunción es la disyunción de las negaciones**, y viceversa — `not (a and b)` equivale a `not a or not b`, y `not (a or b)` a `not a and not b`. Dominar estas leyes permite simplificar condiciones y evitar una de las fuentes más comunes de errores lógicos: negar incorrectamente expresiones compuestas. Véase también (Rosen 2019).

65: Augustus De-Morgan, 1806-1871



Las leyes de De Morgan pueden expresarse así:

$$\neg(P \wedge Q) \equiv (\neg P) \vee (\neg Q)$$
$$\neg(P \vee Q) \equiv (\neg P) \wedge (\neg Q)$$

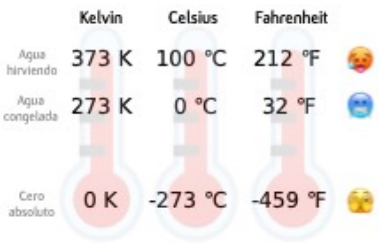
En C++, estas leyes se aplican como:

```
not ( P and Q)  → not P  or not Q
not ( P  or Q)  → not P and not Q
```

Por ejemplo:

```
not ( (A == B) or (A == C) ) → (A != B) and (A != C)
not ( (A == B) and (C > D) ) → (A != B) or (C <= D)
```

66: Termómetros y valores negativos



Las expresiones lógicas que comienzan con una negación son generalmente más difíciles de comprender. Por eso las leyes de De Morgan son especialmente útiles (y se utilizarán a lo largo del libro): permiten reescribir la negación de una condición compleja como una conjunción o disyunción de condiciones más simples y afirmativas. Existe abundante literatura sobre las dificultades cognitivas que experimenta el cerebro al procesar valores negativos. Algunos argumentan que las temperaturas en Fahrenheit son negativas con menos frecuencia que en Celsius y, por ello, resultan más fáciles de comprender. Esta afirmación es muy discutible desde numerosas perspectivas. No obstante, la idea clave es que las negaciones pueden manejarse a menudo con mayor eficacia expresándolas como afirmaciones positivas de negaciones simples. Supongamos que tenemos las variables

```
bool rains = true;
bool carryngUmbrella = false;
```

en nuestro programa y queremos determinar si nos mojaremos. Normalmente nos mojamos si llueve y no llevamos paraguas, por lo que la expresión

```
bool illGetWet = rains and not carryngUmbrella;
```

representa esta condición. La negación de esto sería:

```
bool illNOTGetWet = not (rains and not carryngUmbrella);
// but better after applying De Morgan:
bool illNOTGetWet = not rains or carryngUmbrella;
```

2·11.4 Punto de control

1) Suponiendo

```
x = 3;
y = 6;
z = 8;
```

indica si cada una de las siguientes expresiones relacionales es verdadera o falsa:

2) `4 <= x + z`

3) `y == 3`

4) `x + 3 != y`

5) `z != 4`

6) `x >= 0`

7) `x <= (y * 2)`

8) ¿Son equivalentes las siguientes expresiones relacionales?

   8.a) `x != y` ⇔ `y >= x`

   8.b) `x >= y` ⇔ `y <= x`

   8.c) `x <= y` ⇔ `y > x`

9) ¿Son correctas las siguientes implicaciones (⇒ denota «implica»):

   9.a) `x > y and x < z` ⇒ `y < z` ?

   9.b) `x >= y and z == x` ⇒ `z == y` ?

   9.c) `x != y and x != z` ⇒ `z != y` ?

10) ¿Qué mostrará el siguiente fragmento de programa?

```
int a = 0, b = 2, x = 4, y = 0;
cout << boolalpha;
cout << (a == b) << " " << (a != y) << " "
     << (b <= x) << " " << (y > a) << endl;
```

Tarea 2: Booleanos

Escribe los siguientes programas con los nombres indicados. Esta tarea se centra en valores booleanos, salida formateada con `cout` y aspectos básicos de compilación.

1 Escribe un programa `t0201.booleans.cpp` que imprima el valor de cada una de las siguientes expresiones, o indique que la expresión es inválida si carece de sentido.

```
true and true
false and true
1 == 1 and 2 == 1
1 == 1 or 2 != 1
true and 1 == 1
1 != 0 and 2 == 1
not (true and false)
not (1 == 1 and 0 != 1)
not (10 == 1 or 100 == 100)
not (1 != 10 or 3 == 4)
1 == 1 and not (1 == 1 or 1 == 0)
```

Escribe `cout << boolalpha;` al principio de `main()` y usa el siguiente estilo para cada línea:

```
cout << "true and true -> " << (true and true) << endl;
```

Sin `boolalpha`, `cout` imprime `1` o `0` en lugar de `true` o `false`. Usa la siguiente plantilla: X

```cpp
// 01.booleans.cpp
// juanfc 2026-09-09
#include <iostream>
using namespace std;

int main()
{
    cout << boolalpha;
    cout << "true and true -> " << (true and true) << endl;
    // ...
    cout << "nonsense      -> " << " has no sense" << endl;
    // ...
    return 0;
}
```

2 En C++, las comparaciones encadenadas como `a < b < c` no funcionan como en matemáticas — cada par debe escribirse por separado y unirse con `and`. Escribe un programa `t0202.concatbool.cpp` que pida al usuario un número x e imprima (`true`/`false`) si $0 \leq x \leq 100$. › Sol

3 Dadas las declaraciones X

```
char c;
int x, y;
```

En C++, una expresión booleana como `x > 5` se evalúa como `true` cuando el valor almacenado en `x` es mayor que 5. El objetivo es escribir expresiones booleanas simples que se evalúen como `true` cuando se cumple la condición y como `false` en caso contrario.
Por ejemplo: *«x es par»* → `x % 2 == 0`.

1. `x` termina en `0`,
2. `x` termina en `00`,
3. `x` tiene exactamente 3 dígitos,

4. `x` pertenece a $\{3, 4, 5, 6, 7\}$,
5. `x` pertenece a $\{1, 2, 3, 7, 8, 9\}$,
6. `x` pertenece a $\{1, 2, 3, 7, 8, 9\}$
   y simultáneamente
   `y` pertenece a $\{3, 6, 7, 8\}$,
7. ni `x` ni `y` es mayor que `10`,
8. `c` es una letra mayúscula,
9. `c` es una letra (mayúscula o minúscula),
10. `c` es una vocal minúscula (`'a'`, `'e'`, `'i'`, `'o'`, `'u'`),

4 El programa `t0203.calculator.cpp` mostrado a continuación pide al usuario dos números `float` y un operador (`+ - * /`) e imprime el resultado: › Sol

```cpp
// 03.calculator.cpp
// juanfc 2026-09-09
#include <iostream>
using namespace std;

int main()
{
    float a, b, r;
    char op;

    cout << "Enter 2 numbers: " << endl;
    cin >> a >> b;
    cout << "Enter operation (+, -, *, /): ";
    cin >> op;

    if (op == '+')
        r = a + b;
    else if (op == '-')
        r = a - b;
    else if (op == '*')
        r = a * b;
    else if (op == '/')
        r = a / b;
    else {
        cout << "Error: unknown operator " << op << endl;
        r = 0;
    }

    cout << "r: " << r << endl;
    return 0;
}
```

Extiéndelo para que soporte la exponenciación (`^`), que C++ no proporciona como operador. Incluye `<cmath>` y usa `pow(a, b)`. Para un único valor despachado sobre un tipo ordinal, `switch` es más claro que una cadena de `if`/`else if`.

5 Escribe un programa `t0204.demorgan1.cpp` que pida al usuario dos enteros `x` e `y`, calcule e imprima la expresión booleana `not (x < y and y > 5)`. Calcula e imprime también una expresión equivalente eliminando el `not` mediante las leyes de De Morgan. › Sol

6 Escribe un programa `t0205.demorgan2.cpp` que defina `const int A = 2, B = 3, C = 2`. Partiendo de

```
bool r = not (found and (A==B) or (A==C));
```

reescribe la expresión eliminando el `not` más externo y simplificando mediante las leyes de De Morgan. Imprime `r` con `boolalpha`. › Sol

HINT: Asigna `found = true` como valor concreto. Como regla mnemotécnica, la precedencia de operadores es paralela a la aritmética: `not` (menos unario), `and` (multiplicación), `or` (suma).

o O o

2·12 Estructuras de selección

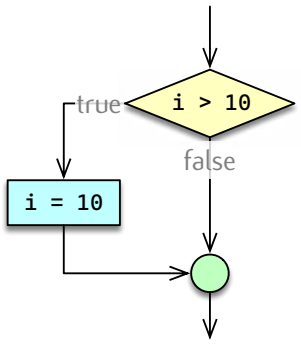
Las estructuras de selección permiten que un programa tome diferentes acciones según una condición, del mismo modo que cogemos el paraguas solo `if` está lloviendo. C++ ofrece dos de ellas, adecuadas a diferentes situaciones.

2·12.1 Estructura if

Puedes imaginar las sentencias de un programa como pasos individuales que das mientras caminas por un camino. Para llegar a tu destino, debes empezar por el principio y dar cada paso, uno tras otro, hasta llegar. Los programas que has escrito hasta ahora siguen un único «camino de ejecución» que el programa debe recorrer.

El tipo de código mostrado hasta ahora se denomina de estructura secuencial porque las sentencias se ejecutan en secuencia, una tras otra, de arriba a abajo, sin desviarse en otra dirección. Sin embargo, los programas frecuentemente requieren más de un camino de ejecución (como se muestra en (Böhm, Jacopini 1966)) para manejar situaciones durante el cómputo o la realización de alguna tarea. Muchos algoritmos requieren que un programa ejecute ciertas sentencias solo bajo determinadas circunstancias. Esto puede lograrse con una estructura de decisión.

67: Condicionar una parte del código



La sentencia `if` es la estructura de control fundamental para tomar decisiones: si se cumple una condición, ejecuta este código. Aunque conceptualmente sencilla, las sentencias `if` anidadas y encadenadas requieren un manejo cuidadoso para mantener la legibilidad del programa. Cuando el programa encuentra una sentencia `if` en su forma simple, evalúa la condición booleana. Si la condición se evalúa como `true`, se ejecuta el *cuerpo* de la primera rama (el código que sigue inmediatamente). Si la condición se evalúa como `false`, ese bloque de código se omite.

```C++
if (n > 10) {
    n = 10;
}
```

Ejemplo:

```
int n = 3;
if (n % 2 == 0) {
    cout << "The number " << n << " is even." << endl;
}
```

2·12.1.1 Punto de control

1) Escribe una sentencia `if` que realice la siguiente lógica: si el valor de la variable `price` es mayor que `100`, asigna `0.2` a la variable `discountRate`.
2) ¿Qué ocurre cuando la condición booleana se evalúa como `false`?
3) ¿Qué ocurre cuando no hay llaves `{ }` alrededor de la siguiente expresión tras la condición booleana?

```
if (n % 2 == 0)
  cout << "The number " << n << " is even." << endl;
```

4) ¿Qué ocurre cuando no hay ninguna expresión tras la condición booleana? Es decir, cuando escribimos

```
if (n % 2 == 0)
  ;
```

5) Escribe un programa completo que lea un `int` edad del usuario e imprima `"You are an adult."` solo cuando `age` sea 18 o mayor. Pruébalo con las edades 17, 18 y 40 para asegurarte de que el mensaje aparece exactamente cuando debe.

6) Escribe un programa completo que lea un `double` temperatura e imprima `"Warning: high temperature!"` solo cuando `temperature` sea superior a 35.0. Pruébalo con 20.0, 35.0 y 40.0.

7) Escribe una sentencia `if` que multiplique `payRate` por `1.5` si `hours` es mayor que `40`.

8) Escribe una sentencia `if` que realice la siguiente lógica: si la variable `sales` es mayor que `50,000`, asigna `0.25` a la variable `commissionRate` y `250` a la variable `bonus`.

9) ¿Qué haría el siguiente código para cualquier valor de `calls`?

```
if (calls = 20)
    rate *= 0.5;
```

10) Escribe la forma más simple de una sentencia que realice la siguiente lógica: si la variable `ticketsSold` es igual a `200`, establece la variable bandera booleana `soldOut` a `true`; en caso contrario, establece `soldOut` a `false`.

11) Escribe la forma más simple de una sentencia que realice la siguiente lógica: si la variable `ticketsSold` es igual a `200`, establece la variable bandera booleana `soldOut` a `true`; en caso contrario, no modifiques `soldOut`.

12) Aunque los siguientes fragmentos de código son sintácticamente correctos, cada uno contiene un error lógico. Localiza el error e indica qué está mal.

A.

```
hours = 12;
if (hours > 40);
   cout << hours << " hours qualifies for over-time." << endl;
```

B.

```
interestRate = .05;
if (interestRate = .07)
   cout << "This account is earning the maximum rate." << endl;
```

C.

```
interestRate = .05;
if (interestRate > .07)
   cout << "This account earns a 10€ bonus." << endl;
balance += 10.0;
```

2·12.1.2 Estructura if-else

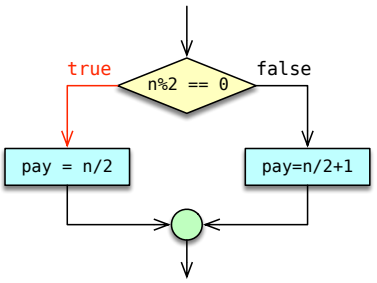
Cuando una condición booleana se evalúa como `false`, la sentencia `if` omite el bloque de código y la ejecución continúa en la siguiente sentencia del programa. Sin embargo, este enfoque puede ser ineficiente, ya que sabemos que la condición es falsa, lo que frecuentemente ofrece

la oportunidad de ejecutar una acción alternativa. En C++, esto se gestiona mediante la rama `else` de la estructura `if`.

Podemos reconocer la falsedad de la condición como una oportunidad para realizar una acción diferente. En sentido estricto, esto también podría lograrse evaluando una condición negada con otra sentencia `if (not cond)` para ejecutar las acciones correspondientes cuando `(not cond)` sea `true`. La rama `else` existe precisamente para aprovechar este conocimiento: cuando sabemos que la condición es falsa, podemos ejecutar código alternativo sin la sobrecarga de reevaluar la condición.

```
int n = 3; // just to see an example
if (n % 2 == 0) {
    cout << "The number " << n << " is even." << endl;
} else {
    cout << "The number " << n << " is odd." << endl;
}
```

68: Estructura if-else. Aprovechamos el conocimiento de que la condición es falsa para realizar una acción alternativa




```
if (n % 2 == 0) {  C++
    pay = n/2;
} else {
    pay = n/2 + 1;
}
```

2·12.1.3 Punto de control

1) Escribe una sentencia `if`/`else` que asigne `0.10` a `commission` a menos que las ventas sean mayores o iguales a `50,000.00`, en cuyo caso asigna `0.20` a `commission`.

2) Escribe una sentencia `if`/`else` que asigne `1` a `x` si `y` es igual a `100`. En caso contrario, debe asignar `0` a `x`.

3) Escribe una sentencia `if`/`else` que asigne `.10` a la variable `discount` si la variable bandera booleana `prepaid` es `true`, y asigne `0.0` a `discount` si `prepaid` es `false`.

4) ¿Funcionará la sentencia `if`/`else` mostrada a la derecha exactamente igual que las dos sentencias `if` separadas mostradas a la izquierda?

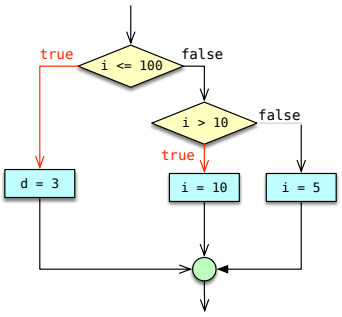
```
if (x < y)                  if (x < y)
    cout << 1;                  cout << 1;
if (x > y)                  else
    cout << 2;                  cout << 2;
```

5) Sigue las instrucciones para completar el siguiente fragmento de programa.

```
enum TPaint { RED, BLUE, YELLOW, GREEN, ORANGE, PURPLE };
TPaint color = GREEN;
```

Escribe una sentencia `if`/`else` que imprima «primary colour» si `color` es `RED`, `BLUE` o `YELLOW`, y que imprima `"mixed color"` en caso contrario. La condición del `if` debe usar una expresión relacional.

69: Podemos continuar evaluando condiciones en la rama else, sabiendo que la condición anterior fue `false`, y responder a cada situación en consecuencia




```
if (i <= 100) {  C++
  d = 3;
} else {
  if (i > 10) {
    i = 10;
  } else {
    i = 5;
  }
}
```

2·12.2 Encadenamiento de estructuras if

Cuando una condición booleana se evalúa como `false`, puedes usar la rama `else` para evaluar condiciones adicionales, sabiendo que la primera condición no se cumplió. Esto permite encadenar múltiples condiciones:

```
int score;
cout << "Enter your score: ";
cin >> score;
if (score >= 9) {
    cout << "Your grading is Sobresaliente" << endl;
} else if (score >= 7) {
```

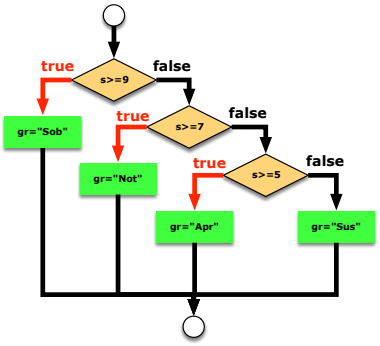
```
    cout << "Your grading is Notable" << endl;
} else if (score >= 5) {
    cout << "Your grading is Aprobado" << endl;
} else {
    cout << "Your grading is Suspenso" << endl;
}
```

Nótese que solo la rama final puede omitir una condición booleana; se ejecuta cuando todas las condiciones anteriores se han evaluado como `false`.

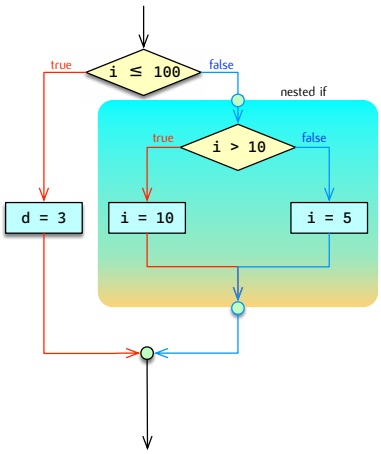
El orden en que se especifican las condiciones es crítico, ya que cada nueva condición solo se evalúa si todas las condiciones anteriores se han evaluado como `false`.

En el ejemplo anterior, utilizamos el encadenamiento if-else para filtrar valores comprobando repetidamente la variable `score`. Este enfoque no solo

70: Encadenamiento de preguntas



71: Dentro de cada rama, conociendo el resultado que llevó a ese punto, podemos plantear nuevas preguntas mediante if



2·12.3 Sentencias if anidadas

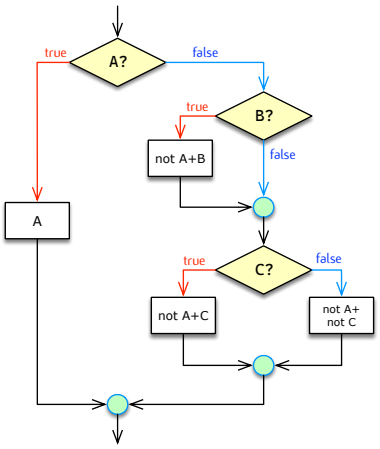
Hemos visto que las respuestas negativas a preguntas anteriores pueden aprovecharse encadenando nuevas preguntas. Sin embargo, también es posible plantear preguntas adicionales con independencia de las respuestas anteriores, anidando la pregunta dentro de las ramas correspondientes de la estructura if-else.

En Figura 71, sabiendo que la respuesta a la pregunta exterior es `false`, podemos formular una pregunta adicional para decidir entre dos procesos alternativos.

En Figura 72 podemos ver diferentes situaciones lógicas en las que puede encontrarse cada rama.

La diferencia entre un if-encadenado y un if-anidado es que en este último las situaciones pueden obtenerse combinando respuestas positivas y negativas de preguntas anteriores, mientras que en un if-encadenado cada respuesta refina únicamente una situación lógica, creando una forma de filtrado lógico.

72: Situaciones en las que cada rama depende de respuestas previas



2·12.3.1 Punto de control

1) Si ejecutas el siguiente código, ¿qué mostrará si el usuario introduce 5? ¿15? ¿30? ¿–1?

```
cout << "Enter a number: ";
cin >> number;
if (number > 0) {
  cout << "Zero    ";
  if (number > 10) {
    cout << "Ten    ";
    if (number > 20) {
      cout << "Twenty             ";
    }
  }
}
```

2) Si ejecutas el siguiente código, ¿qué mostrará si el usuario introduce `15 18`? ¿`15 10`? ¿`9 7`?

```
cout << " Enter the number of team wins and number of team losses: ";
cin >> team Wins >> teamLosses;

if (teamWins > teamLosses)
{
  if(teamWins > 10)
     cout << "You are the champions. " << endl;
  else
     cout << "You have won more than 50% of your games. " << endl;
}
else
  cout << "Good luck in the rest of your games. ";
```

3) ¿Cuál es la forma más general de combinar `if`s: `if`s encadenados o `if`s anidados?

4) ¿Qué mostrará el siguiente fragmento de programa?

```
int funny = 1, serious;
if (funny != 1) {
  funny = serious = 1;
} else if (funny == 2) {
  funny = serious = 3;
} else {
  funny = serious = 5;
}
cout << funny << " " << serious << endl;
```

5) El siguiente programa se usa en una librería para determinar cuántos cupones de descuento recibe un cliente. Completa la tabla que aparece tras el programa.

```
#include <iostream>                                                    C++
using namespace std;
int main()
{
  int numBooks, numCoupons;

  cout << "How many books are being purchased? ";
  cin >> numBooks;
  if (numBooks < 1)
     numCoupons = 0;
  else if (numBooks < 3)
     numCoupons = 1;
  else if (numBooks < 5)
     numCoupons = 2;
  else
     numCoupons = 3;

  cout << "The number of coupons to give is " << numCoupons << endl;

  return 0;
}
```

¿Cuántos cupones recibirá el cliente tras comprar 1, 2, 3, 4, 5, 10 libros?

6) Escribe una sentencia `if`/`else if` que lleve a cabo la siguiente lógica. Si el valor de la variable `quantityOnHand` es igual a `0`, muestra el mensaje `"Out of stock"`. Si el valor es mayor que `0` pero menor que `10`, muestra el mensaje `"Reorder"`. Si el valor es `10` o más, no muestres nada.

7) Escribe una sentencia `if`/`else if` que realice las mismas acciones que en la pregunta anterior cuando el valor de `quantityOnHand` es igual a

`0` o es mayor que `0` pero menor que `10`. Sin embargo, cuando el valor sea `10` o más, debe mostrar el mensaje `"Quantity OK"`.

2·12.3.2 Decisión ternaria

Aunque no se usará de forma extensiva, puede servir como atajo para reducir la verbosidad del código. Como dato de interés, es el único operador ternario en C, es decir, implica tres parámetros en su evaluación:

Decisión normal	Decisión ternaria

`int max;`
`if (a > b)`
`  max = a;`
`else`
`  max = b;`
	`int max = a > b ? a : b;`

2·12.3.3 Ámbitos creados por bloques

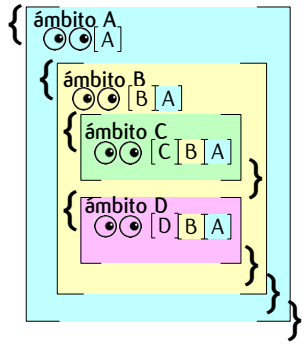
En C++, una llave de apertura `{` introduce un nuevo ámbito, mientras que `}` lo cierra. Esto establece una capa anidada en la que se puede acceder a todas las entidades accesibles en el ámbito envolvente, así como a cualquier nueva variable, tipo y otros elementos del lenguaje declarados dentro del bloque. Cuando el bloque termina, todas las entidades declaradas en su interior son destruidas y dejan de ser accesibles.

```
int a = 10;
cout << "Outside a: " << a << endl;
{
  int a = 33;
  cout << "Inside a: " << a << endl;
}
cout << "Outside, after block, a: " << a << endl;
```

¿Qué produciría esta salida?

Más adelante examinaremos las funciones. El aspecto más importante de las funciones es que crean bloques y ámbitos propios, independientes del código circundante. Este aislamiento garantiza que es seguro trabajar con variables y definiciones declaradas dentro de ellas, ya que se sabe que estarán ocultas y no interferirán con el resto del programa.

73: Los ámbitos internos pueden ver los que los envuelven, pero no los adyacentes ni los que están dentro de ellos



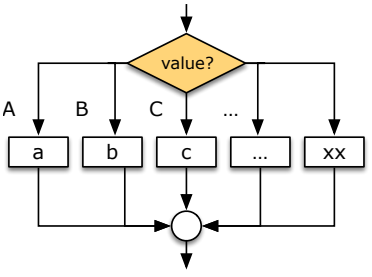
2·12.4 Estructura switch

Cuando una decisión depende de comparar un único valor ordinal con varias alternativas conocidas, la estructura `switch` proporciona mayor claridad que una cascada de sentencias `if`. Recuerda que los tipos ordinales incluyen `bool`, `char`, `int` y sus derivados. También examinamos su característico comportamiento de caída entre casos.

```
switch (achar) {
  case 's':
     y = sin(x);
     break;
  case 'c':
     y = cos(x);
     break;
  // ...
  default:
     cerr << "Error" << endl;
}
```

```
switch (value) {
case A:
  // a
break;
case B:
  // b
break;
// ...
default:
  // xx
}
```

74: La estructura de control switch es sencilla en C++ y especialmente conveniente cuando un valor escalar sirve como clave de acceso a varias acciones diferentes



Las etiquetas pueden sucederse sin sentencia `break`, apilando los `case ..:`s, y esta es la única manera que ofrece C++ de usar varias claves diferentes (no demasiadas) para la misma acción.

```
switch (achar) {
  case 's':
  case 'S':
     y = sin(x);
     break;
  case 'c':
  case 'C':
     y = cos(x);
     break;
  // ...
  default:
     cerr << "Error" << endl;
}
```

La cláusula `default`, si está presente, debe ser la última y es la única que no requiere sentencia `break`.

Una aplicación fundamental de `switch` es la implementación de menús:

```
cout << "0: Exit" << endl;
cout << "1: Do something" << endl;
cout << "2: Do another thing" << endl;
cout << "3: etc" << endl;
int selection;
cout << "Please, user, choose an option: ";
cin >> selection;

switch (selection) {
 case 0:
   return 0; // THE END of the program
 case 1:
    // Do something
 break;
 case 2:
    // Do another thing
 break;
 case 3:
    // Do etc
 default:
    cout << "Invalid option" << endl;
 break;
}
```

Por ejemplo, si estamos implementando una calculadora:

```
float a, b, r;

cout << "Enter 2 numbers: " << endl;
cin >> a >> b;

cout << "Operation: ";
char op;
cin >> op;
switch (op) {
 case '+':
    r = a+b;
    break;
 case '-':
    r = a-b;
    break;
 case '*':
    r = a*b;
    break;
 case '/':
    r = a/b;
    break; // This one isn't necessary.
}
```

```
cout << "r: " << r << endl;
```

Normalmente querremos repetir el menú al usuario:

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

int main()
{
    int option;
    do {
        cout << "Enter 0 to exit" << endl;
        cout << "Enter 1 to show a message" << endl;
        cout << "Enter 2 to do something else" << endl;

        cout << "Your option?: ";
        cin >> option;
        switch (option) {
        case 1:
            cout << "The message" << endl;
        break;
        case 2:
            cout << "Another thing" << endl;
        break;
        }
    } while (option != 0);

    return 0;
}
```

He aquí otro ejemplo de sentencia `switch`. La función `toupper(grade)` convierte y devuelve `grade` en mayúscula si originalmente estaba en minúscula[1]:

[1] Lo implementaremos como ejemplo, pero ya está definida en `<cctype>`, por lo que para usarla hay que recordar incluir `#include <cctype>` al principio del programa.

```
char grade;
cout << "Enter your control grade (A, B, C, D, E or F): ";
cin >> grade;

switch (toupper(grade)) {
case 'A':
    cout << "Excellent. "
         << "You need not take the final." << endl;
    break;

case 'B':
    cout << "Very good. ";
    grade = 'A';
    cout << "Your midterm grade is now "
         << grade << endl;
    break;

case 'C':
    cout << "Passing." << endl;
    break;

case 'D':
case 'F':
    cout << "Not good. "
         << "Go study." << endl;
    break;

default:
    cout << "That is not a possible grade." << endl;
}
```

2·12.4.1 Punto de control

1) Explica por qué no puedes convertir la siguiente sentencia `if`/`else if` en un `switch`.

```
if (temp == 100)
   x = 0;
else if (population > 1000)
   x = 1;
else if (rate < .1)
   x = -1;
```

2) ¿Qué está mal en la siguiente sentencia `switch`?

```
switch (temp)
{
  case temp < 0 :
    cout << "Temp is negative." << endl;
  break;
  case temp == 0:
    cout << "Temp is zero." << endl;
  break;
  case temp > 0 :
    cout << "Temp is positive." << endl;
  break;
}
```

3) ¿Qué mostrará el siguiente fragmento de programa?

```
int funny = 7, serious = 15;
funny = serious * 2;
switch (funny)
{ case 0 : cout << "That is funny." << endl;
           break;
  case 30: cout << "That is serious." << endl;
           break;
  case 32: cout << "That is seriously funny." << endl;
           break;
  default: cout << funny << endl;
}
```

4) Completa el siguiente fragmento de programa escribiendo una sentencia `switch` que muestre `"one"` si el usuario introduce `1`, `"two"` si introduce `2`, y `"three"` si introduce `3`. Si se introduce un número distinto de `1`, `2` o `3`, el programa debe mostrar un mensaje de error.

```
cout << "Enter one of the numbers 1, 2, or 3: ";
cin >> userNum;

// Write the switch statement here.
```

5) Reescribe el siguiente fragmento de programa usando una sentencia `switch` en lugar de la sentencia `if`/`else if`.

```
int selection;
cout << "Which formula do you want to see?" << endl<<endl;
cout << "1. Area of a circle" << endl;
cout << "2. Area of a rectangle" << endl;
cout << "3. Area of a cylinder" << endl;
cout << "4. None of them!" << endl;
cin >> selection;
if (selection == 1)
  cout << "Pi times radius squared" << endl;
else if (selection == 2)
  cout << "Length times width" << endl;
else if (selection == 3)
  cout << "Pi times radius squared times height" << endl;
else if (selection == 4)
  cout << "Well okay then, good-bye!" << endl;
else
  cout << "Not good with numbers, eh?" << endl;
```

Tarea 3: Estructuras de control. Condicionales

Resuelve cada problema dentro de `main()` usando condicionales. No se permiten estructuras de datos — las veremos más adelante.

1 Responde a estas preguntas en un único fichero `t03e01.questions.cpp` etiquetando cada una con su letra. Por ejemplo (¡estas son respuestas incorrectas!): › Sol

```
// a) P4 se ejecuta cuando a == 33 y b == 0                    display
// b) a > b and b == 10
```

a) Considera el siguiente fragmento de código, donde P_i $(i = 1, 2, 3, 4)$ son segmentos de código arbitrarios:

```
if      (a >  0) P1;
else if (b != 0) P2;
else if (a == b) P3;
else if (a == 0) P4;
```

¿Para qué valores de `a` y `b` se ejecuta P_4?

b) Suponiendo que `a` y `b` son enteros, escribe una expresión booleana simple que sea `true` cuando ninguno de los dos sea mayor que `10`.

c) Simplifica el siguiente fragmento de código:

```
if (cond) {
    T;
    S;
} else {
    T;
    U;
}
```

d) Simplifica el siguiente fragmento de código:

```
if (cond)
    x = true;
else
    x = false;
```

e) Simplifica el siguiente fragmento de código usando el menor número posible de condiciones. Usa las palabras reservadas `and` `or` `not` cuando sea necesario:

```
if (age >= 65) {
    state = RETIRED;
} else {
    if (age < 18) {
        state = MINOR;
    } else {
        if ((age >= 18) and (age < 65)) {
            state = ACTIVE;
        }
    }
}
```

f) Examina los siguientes pares de expresiones y determina si son equivalentes. Para cada par, responde con `true` o `false`.

1: `not (a == b)` $\overset{?}{\Leftrightarrow}$ `a != b`
2: `not ((a==b) or (a==c))` $\overset{?}{\Leftrightarrow}$ `(a!=b) and (a!=c)`
3: `not ((a==b) and (c>d))` $\overset{?}{\Leftrightarrow}$ `(a!=b) or (c<=d)`
4: `not (a and b)` $\overset{?}{\Leftrightarrow}$ `a or b`
5: `a and b or c` $\overset{?}{\Leftrightarrow}$ `a and (b or c)`
6: `a and b or c` $\overset{?}{\Leftrightarrow}$ `not (a or b and c)`

2 Escribe un programa `t03e02.prices.cpp` que calcule el precio de un lote de artículos. El precio unitario disminuye con la cantidad según la siguiente tabla (la cantidad se lee por teclado): › Sol

Cantidad	Precio unitario
1	100
2	95
3	90
>= 4	85

El programa pedirá la cantidad y mostrará el precio total. Por ejemplo:

```
Quantity: 3
Total: 270€
```
display

3 Construye un programa `t03e03.divisibleby7.cpp` que lea un entero `n` del usuario y determine si es divisible por 7 usando este algoritmo:
1. duplica el último dígito de `n` y réstalo del número restante;
2. si el resultado es divisible por 7 (o igual a `0`), el número original es divisible por 7.

Por ejemplo: para `n = 203`, duplica el último dígito ($3 \times 2 = 6$) y réstalo de la parte restante ($20 - 6 = 14$). Como `14` es divisible por 7, `203` es divisible por 7. › Sol

4 Construye el programa `t03e04.largest4.cpp` que pide al usuario 4 enteros y muestra cuál es el mayor. No uses más tipos de los vistos en clase: tipos simples. › Sol

5 Escribe un programa `t03e05.quadraticeq.cpp` para resolver ecuaciones de segundo grado ($ax^2 + bx + c = 0$) pidiendo los coeficientes (a, b y c) y aplicando la fórmula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

teniendo en cuenta: › Sol › complete

1. no intentar calcular la raíz cuadrada de un discriminante negativo ($d = b^2 - 4ac$),
2. no dividir por cero (si el usuario introduce un coeficiente a igual a cero),

Informa al usuario cuando se produzca cualquiera de estas dos condiciones. Intenta resolverlo (🧊) cubriendo las seis situaciones posibles: dos raíces reales distintas, una raíz real repetida, raíces complejas conjugadas, coeficiente a igual a cero con b distinto de cero, todos los coeficientes iguales a cero y solo c distinto de cero.

6 Diseña un programa `t03e06.timedecomposition.cpp` que pida un número (`int`) de segundos e imprima el número de días, horas, minutos y segundos que componen esa cantidad. Por ejemplo, `5000` segundos deberían dar 1 hora, 23 minutos y 20 segundos. › Sol

7 Escribe un programa `t03e07.dogage2.cpp` para calcular la edad de un perro en años equivalentes humanos. Pide la edad del perro en años y su peso en kg, luego aplica la siguiente regla: › Sol

a) **Pequeño** (< 10 kg): los primeros 2 años = 12,5 años humanos cada uno; cada año posterior = 4 años humanos

b) **Mediano** (10–25 kg): los primeros 2 años = 10,5 años humanos cada uno; cada año posterior = 4 años humanos

c) **Grande** (> 25 kg): los primeros 2 años = 9 años humanos cada uno; cada año posterior = 5 años humanos.

Rechaza entradas no positivas. Imprime el resultado redondeado al entero más próximo.

```
Enter dog's age in years: 15                                    display
Enter dog's weight in kg: 30
The dog's age in human years is 83
```

HINT: Una fórmula más precisa (pero no lineal) basada en la metilación del ADN fue publicada por Wang & Ideker, *Cell Systems*, 2020: $\text{human age} = 16\ln(\text{dog age}) + 31$. Ver doi:10.1016/j.cels.2020.06.006.

8 Escribe un programa `t03e08.astro.cpp` que muestre el signo astrológico para una fecha de nacimiento dada. Ejemplo: › Sol

```
Enter the day of birth: 15                                      display
Enter the month of birth (e.g. March, July, etc.): May
Your Astrological sign is: Taurus
```

HINT: Para consultar las fechas de los signos del zodiaco occidental, ver aquí.

9 Escribe un programa `t03e09.chinesezodiac.cpp` que muestre el signo del zodiaco chino para un año de nacimiento dado. Toma la lista año–signo de aquí. › Sol

```
Enter your year of birth: 2023                                  display
Your Zodiac sign: Rabbit
```

10 Escribe el programa `t03e10.chartype.cpp` que lee un `char` del usuario e imprime uno de los siguientes mensajes: «es un dígito», «es una vocal minúscula», «es otra cosa». › Sol

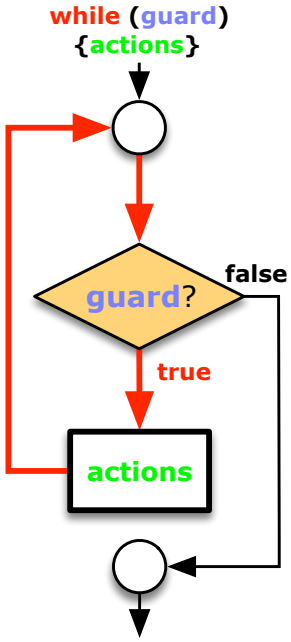
2·13 Estructuras de iteración

Mientras que las estructuras condicionales nos permiten añadir lógica y flexibilidad a nuestros programas, las estructuras de iteración nos permiten especificar acciones que se repiten sin necesidad de escribir código explícito para cada ocurrencia.

> Los condicionales dan inteligencia a nuestros programas,
> mientras que la iteración les da potencia

Los ordenadores destacan en la repetición: ejecutar una operación un millón de veces es computacionalmente trivial. Los bucles son las estructuras que expresan la iteración, y C++ proporciona dos variantes que difieren en dónde y cómo se evalúa la condición de continuación. También ofrece una variante adicional diseñada para el recorrido de rangos, una operación tan frecuente en programación que merece su propia construcción de bucle. Aquí tratamos los bucles `while`, `do-while` y `for` (para recorrido de rangos).

75: El bucle while puede no ejecutar nunca las acciones de su cuerpo; la condición guarda puede impedir la ejecución incluso una vez



2·13.1 Estructura while

El bucle `while` repite un bloque de código mientras una condición sea verdadera, evaluando la condición **antes** de cada iteración — por consiguiente, el cuerpo puede no ejecutarse nunca. Es la construcción de bucle más general, de la que se pueden derivar todas las demás.

En el siguiente ejemplo, observa que debemos obtener un valor inicial para `x` antes de entrar (o decidir no entrar) en el **cuerpo** de las acciones que el bucle `while` repetirá. Tras cada iteración, el programa solicita al usuario un nuevo valor de `x`. El programa termina cuando el usuario proporciona un valor de `0` o menor (ya que es imposible calcular $\sqrt{x}$ para valores no positivos).

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

int main()
{
    float x;

    cout << "Square root of (0 or < 0 ends): ";
    cin >> x;
    while ( x > 0) {
        cout << sqrt(x) << endl;
        cin >> x;
    }

    return 0;
}
```

Una variante práctica del programa anterior, que resultará especialmente útil dado que solicitar valores al usuario será frecuente en nuestros ejercicios, se muestra a continuación:

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

```
int main()
{
    float x;

    cout << "Square root of (0 or < 0 ends): ";
    while (cin >> x and x > 0) {
        cout << sqrt(x) << endl;
    }

    return 0;
}
```

Observa que hemos omitido el mensaje de entrada separado. La operación de entrada (`cin >>`) se realiza simultáneamente con la evaluación de la condición principal del bucle (`x > 0`). Este enfoque es posible porque `cin` devuelve `true` cuando la operación de lectura tiene éxito.

2·13.1.1 Punto de control

1) ¿Qué mostrará la ejecución de las siguientes líneas de código?

```
int count = 1;
while (count < 10) {
  cout << count << " ";
  count = count + 2;
}
```

2) ¿Qué hace el siguiente programa?

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

int main()
{
    int i = 0;
    while (i < 5) {
        cout << '*';
        ++i;
    }
    return 0;
}
```

3) Escribe un programa que solicite repetidamente al usuario números reales hasta que el usuario introduzca `0`. Se permiten números negativos. Antes de que comience el bucle, declara una variable `sum` e inicialízala a `0`, luego suma cada valor introducido a ella. Cuando el programa termine, muestra la suma total de todos los números introducidos por el usuario.

4) En el siguiente segmento de programa, ¿qué variable es el contador y cuál es el acumulador?

```
int number, maxNums, x = 0, t = 0;
cout << "How many numbers do you wish to enter? ";
cin >> maxNums;
while (x < maxNums) {
   cout << "Enter the next number: ";
   cin >> number;
   t += number;
   x++;
}
cout << "The sum of those numbers is " << t << endl;
```

5) Encuentra cuatro errores en el siguiente código que intenta sumar cinco números.

```
int val, count, sum;
cout << "I will add up 5 numbers. " << endl

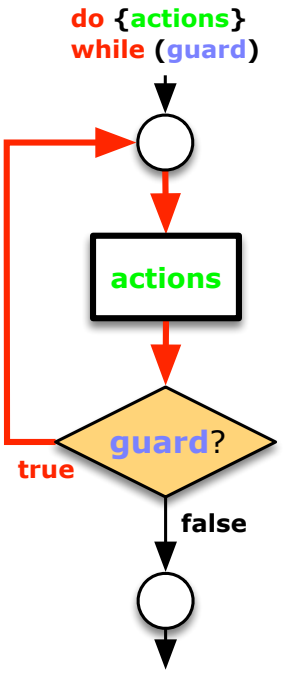
while (count < 5) {
  cout << "Enter a number: ";
  cin >> val;
  sum = val;
}
cout << "The sum of these numbers is " << sum << endl;
```

2·13.2 Estructura do-while

El bucle `do-while` comprueba su condición **después** del cuerpo, garantizando al menos una ejecución. Es la elección natural cuando algo debe ocurrir antes de poder decidir si repetir — por ejemplo, validar la entrada del usuario.

Aunque los bucles `do-while` son menos frecuentes en los programas habituales, resultan especialmente útiles cuando una acción debe realizarse al menos una vez antes de poder evaluar la condición guarda. Esto los hace muy adecuados para situaciones en las que el cuerpo del bucle necesita ejecutarse antes de que pueda aplicarse ninguna lógica de decisión.

76: El bucle do-while es menos frecuente pero resulta conveniente en ciertos casos



Considera un juego en el que el usuario debe adivinar un número secreto conocido únicamente por el programa. En este caso, solicitar números al usuario debe terminar en cuanto introduzca el número correcto. Sin embargo, dado que la comparación entre el número introducido y el secreto solo puede realizarse tras recibir la primera entrada, el bucle `do-while` es la elección apropiada:

```
// guessit.cpp                                                    C++

#include <iostream>
using namespace std;

int main() {
    int secret = 15; // not seen by the user
    int guess;
    do
        cout << "Guess our number: ";
    while (cin >> guess and guess != secret);
    cout << "You got it!" << endl;

    return 0;
}
```

Ahora conviértelo en un programa con bucle `while` para comparar cuál de los dos enfoques es más simple y legible.

2·13.2.1 Punto de control

1) ¿Cuántas veces se ejecutarán las acciones P en la siguiente iteración?

```
cnt = 0;
```

```
while (cnt < 10) {
  cnt++;
  if (cnt == 10)
    P;
}
```

2) En función de `aValue`, ¿cuál es el valor de la variable booleana test tras las siguientes sentencias?

```
bool test = true;
int n = aValue;
int cnt = 1;

while (cnt++ <= n)
    test = not test;
```

3) ¿Qué mostrarán los siguientes fragmentos de programa?

A.

```
int count = 3;
do {
  cout << "Hello World" << endl;
  count—;
} while (count < 1);
```

B.

```
int val = 5;
do {
    cout << val << " ";
} while (val >= 5);
```

C.

```
int count = 0, number = 0, limit = 4;
do {
    number += 2;
    count++;
} while (count < limit);
cout << number << " " << count << endl;
```

2·13.3 Estructura for

Cuando se conoce de antemano el número de iteraciones, el bucle `for` consolida la inicialización, la condición y la actualización en una sola línea. Es la forma idiomática de realizar conteos en C++.

La cabecera de un bucle `for` consta de tres partes: `( ① ; ② ; ③ )` seguidas del cuerpo que contiene las `acciones`.

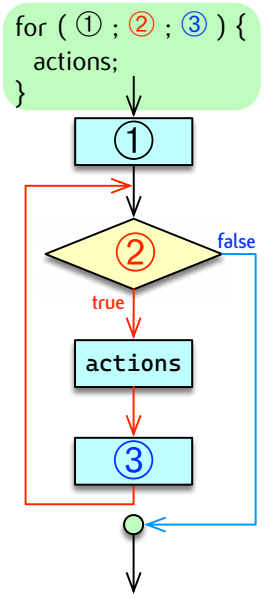
① Se ejecuta una sola vez, antes de que ocurra cualquier otra cosa; siempre se ejecuta. A continuación comienza el bucle propiamente dicho.

② Se comprueba antes de cada ejecución de las `acciones`, de forma similar a los bucles `while`.

③ Se ejecuta después de las `acciones` y antes de la siguiente comprobación de ②.

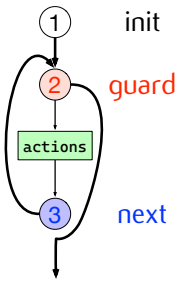
Los bucles `for` son tan cómodos que con frecuencia se usan en exceso, aplicándolos en situaciones en las que no deberían usarse. C++ es bastante permisivo a la hora de exigir el uso correcto de sus elementos y permite escribir cualquier cosa en cada parte, ①, ② y ③, de modo que muchos programadores tienden a escribir expresiones booleanas complejas en ②

77: Un bucle for consta de tres partes: (① ; ② ; ③) más el cuerpo de acciones




```
for ( 1; 2; 3 ) {
  actions
}
```

79: Tres partes + acciones



y pierden el sentido de la propia palabra **for**: iterar a lo largo de un rango. El punto clave para el uso correcto de los bucles `for` es: ¿conozco de antemano el rango de la iteración, o es desconocido el número de veces o el punto que debe cubrir la iteración? Si no sabemos de antemano, antes de que comience el bucle, el número de veces que tiene que iterar (algo que puede depender de variables o constantes previas), entonces el bucle que debemos usar es `while`.

En un bucle `for` típico

```
for (int i = 0; i < N; ++i) {
    // actions
}
```

conocemos de antemano el punto de inicio y el punto final y, no menos importante, modificamos el contador **solo** en la cabecera, nunca en el cuerpo de acciones. Por tanto, con solo mirar la cabecera de un bucle `for`, debemos ser capaces de conocer el número de iteraciones.

Otros aspectos importantes respecto a esta estructura de iteración especial pero de gran relevancia:

punto y coma Ten cuidado de que las partes (① ; ② ; ③) de la cabecera se separan con `;`, no con comas; puedes usar comas dentro de cualquiera de ellas, como permite la sintaxis, por ejemplo en `for (int i = 0, j = 1; i < N and j < M; ++i, ++j)` donde el `;` marca el fin de cada parte de la cabecera.

ámbito Aunque sintácticamente la cabecera está fuera del ámbito de las llaves, de hecho el ámbito de la cabecera (① ; ② ; ③) es el interior de las llaves que encierran las acciones. Por tanto, la declaración `int i = 0` en los ejemplos anteriores declara una variable dentro de las llaves del cuerpo de acciones. Esto es muy útil, ya que puedes mantener intactas las variables externas `i`, y las acciones siempre harán referencia a la «`i` interna». Esto hace que los bucles `for` sean mucho más independientes del contexto circundante, interactuando con él únicamente a través de las acciones específicas que realizas.

Recuerda comprobar que estás usando este bucle correctamente verificando que puedes conocer el número de iteraciones a partir de las variables y constantes involucradas en la cabecera.

2·13.3.1 Punto de control

1) ¿Cuál es el problema del siguiente bucle `for`?

```
for (int i = 0; i < N; ++i) {
    s += i*i;
}
cout << i << endl;
```

2) ¿Está garantizado que el cuerpo del bucle `for` se ejecutará?

3) ¿Podemos usar más de un contador o índice en un único bucle `for`?

4) ¿Cómo se pueden realizar múltiples acciones en la tercera sección de la cabecera de un bucle `for`?

5) ¿Qué hace el siguiente ejemplo?

```
for (int i = 0; i < 80; ++i)
  cout << "*";
```

6) Calcula e imprime la suma de todos los números del 1 al 1.000 usando un bucle `for` simple.

7) El factorial es la «acumulación de productos» de los números naturales hasta el número dado. Es decir: $n! = 1 \times 2 \times 3 \times \ldots \times n$. Escribe un programa con un bucle `for` para calcular el factorial de 7. Ten cuidado con la inicialización del resultado: si lo inicializas a 0, todos los productos se colapsan a 0. Inicialízalo a 1.

8) ¿Y qué hace el siguiente?

```
int s=0;
for ( int i = 1; i <= 100; ++i ) {
  s += i*i;
}
cout << s;
```

2·13.4 Diseño de bucles. Concepto de invariante

Escribir un bucle que funcione es fácil; escribir uno que **probablemente** funcione es una disciplina. El invariante de bucle — una propiedad que es verdadera antes y después de cada iteración — es la herramienta clave para diseñar bucles correctos.

Hablamos de los invariantes de bucle en la sección Sección 1·4.4.1 para comprender cómo analizar y verificar la corrección lógica del código. Un invariante de bucle es una afirmación sobre tus variables que es verdadera inmediatamente antes de que comience el bucle (se **establece**) y sigue siendo verdadera al final de cada iteración (se **mantiene**). El patrón general es el siguiente:

```
// invariant holds here
for ( ... ; TEST CONDITION ; ... ) {
  // ...
  // invariant holds here again
}
// invariant + termination = goal
```

Hay que comprobar dos cosas por separado: por qué termina el bucle, y por qué alcanzar la terminación, junto con el invariante, da el objetivo:

$$\text{invariante} + \text{terminación} \Rightarrow \text{objetivo}$$

Ejemplo. Suma de los primeros `N` números naturales:

```
int sum = 0;
for (int i = 1; i <= N; ++i) {
  // invariant: sum == 1 + 2 + ... + (i - 1)
  sum += i;
  // invariant: sum == 1 + 2 + ... + i
}
// termination: i == N + 1
// invariant + termination: sum == 1 + 2 + ... + N
```

Cada pasada restaura el invariante; una vez que el bucle termina, el invariante más la terminación dan exactamente el objetivo.

2·14 Gestión de errores y excepciones

Las cosas salen mal: los usuarios escriben letras donde se esperan números, los valores caen fuera del rango que una fórmula puede aceptar, una división encuentra un cero. Los programas robustos anticipan estas situaciones, y C++ proporciona mecanismos — desde comprobar el estado de un flujo de entrada hasta las excepciones propiamente dichas — para detectarlas y reaccionar con gracia en lugar de bloquearse.

Aquí no estamos construyendo aplicaciones de producción, por lo que los fallos de los que nuestros programas deben preocuparse son mucho más limitados que en un sistema real. Nuestros programas se ejecutan en un entorno controlado: todavía no estamos leyendo ficheros ni obteniendo datos de internet, y en general podemos esperar que el usuario coopere. Aun así, «en general» no es «siempre» — si pides un número para calcular una raíz cuadrada, un usuario descuidado o curioso puede perfectamente escribir uno negativo. Cuando eso sucede y nada en el programa está vigilándolo, el programa no falla con elegancia: el sistema simplemente lo aborta, normalmente con un mensaje que poco significa para quien está al teclado.

Por tanto, en situaciones que podemos prever, vale la pena proteger el programa y mantenerlo receptivo en lugar de dejar que se bloquee. El enfoque más sencillo es anticipar qué puede salir mal y, cuando ocurra, informar al usuario y quizá permitirle intentarlo de nuevo. Considera:

```
try {
  💥
}
catch (..) {
  ..
}
```

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

int main()
{
    int n, d;
    cout << "Enter numerator: " << endl;
    cin >> n;
    cout << "Enter denominator: " << endl;
    cin >> d;

    cout << n << "/" << d << " = " << n/d << endl;
    return 0;
}
```

Esto se bloquea si el usuario introduce `0` como denominador. La solución es una única guarda:

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

int main()
{
    int n, d;
    cout << "Enter numerator: " << endl;
    cin >> n;
    cout << "Enter denominator: " << endl;
    cin >> d;

    if (d == 0) {
        cout << "ERROR: Impossible to divide by 0" << endl;
    } else {
        cout << n << "/" << d << " = " << n/d << endl;
    }
    return 0;
}
```

Podemos ir más lejos y, en lugar de simplemente negarnos a dividir, dar al usuario otra oportunidad. Este es exactamente el caso anticipado antes en este capítulo▲: leer la entrada es en sí mismo una fuente de errores, ya que un usuario puede escribir una letra donde se espera un número. Un bucle `do-while` — que comprueba la entrada **después** de intentar leerla — combinado con `cin.clear()` y `cin.ignore()` para recuperarse de una lectura fallida, nos permite seguir preguntando hasta obtener algo utilizable:

▲Ver la discusión del estado de fallo de `cin` en Sección 2·9, donde conocimos por primera vez `if (not cin)`.

```cpp
// readint.cpp
#include <iostream>
using namespace std;

int main()
{
    int n;
    bool ok;
    do {
        cout << "Enter an integer: ";
        cin >> n;
        ok = bool(cin);
        if (not ok) {
            cin.clear();              // step out of the fail state
            cin.ignore(1000, '\n');  // discard the bad input still
            waiting
            cout << "That was not a number, try again." << endl;
        }
    } while (not ok);

    cout << "Thanks, you entered: " << n << endl;
    return 0;
}
```

Observa que aquí no hay nada nuevo: `if`, `do-while` y `cin` son construcciones que ya conoces. Gestionar bien los errores tiene menos que ver con sintaxis especial y más con el hábito de preguntarse, en cada punto en que los datos entran en el programa, «¿y si esto no es lo que espero?»

Para situaciones genuinamente impredecibles — donde no podemos simplemente proteger un único valor con un `if`, porque el fallo puede ocurrir varios niveles de llamadas dentro de código que no escribimos nosotros — C++ y la mayoría de los demás lenguajes ofrecen un mecanismo dedicado llamado **excepciones**: un bloque `try` marca una región de código como sensible, y un bloque `catch` separado indica qué hacer si algo dentro de ella falla, de modo que el control salta directamente desde el punto de fallo hasta el manejador correspondiente, saltándose todo lo que hay entre medias. Ese mecanismo solo merece la pena una vez que nuestros programas están divididos en funciones que llaman a otras funciones, que es exactamente adonde se dirige el próximo capítulo — por lo que dejamos `try`/`catch` para más adelante y, por ahora, nos apoyamos en las comprobaciones `if` y `do-while` anteriores.

2·14.1 Punto de control

1) ¿Cuándo es apropiado usar el mecanismo `try`/`catch`/`throw`, y cuándo bastaría con una simple comprobación con un mensaje de error?
2) ¿Son visibles y controlables por nuestros programas todos los errores que genera el sistema?

2·15 Recomendaciones y errores frecuentes

La experiencia muestra que los principiantes se encuentran habitualmente con los mismos escollos: `=` en lugar de `==`, puntos y coma olvidados y errores de uno en uno en los bucles. Este capítulo concluye con una colección de buenas prácticas y errores frecuentes, para que puedas aprender de la experiencia de los demás y no solo de la propia.

A lo largo de esta parte hemos cubierto los pasos para resolver problemas, la elaboración de diagramas, la verificación de bucles mediante sus invariantes, y el uso correcto de los tipos y lo delicado que puede ser mezclarlos. Hemos tratado las particularidades de los enteros sin signo y las complejidades de los números en coma flotante, así como los fundamentos del desarrollo de la lógica mediante condicionales y bucles. En los condicionales, el orden en que se evalúan las condiciones es fundamental; en los bucles, la especialización de los bucles `for`, que con frecuencia se usan mal (quizás debido a su facilidad de escritura y las plantillas rápidas que los editores proporcionan con tanta facilidad), es igualmente importante.

El uso correcto del bucle adecuado — especialmente la idea de no saber *a priori* cuándo terminarán las iteraciones (`while`), frente a conocer el número de iteraciones a partir de la información disponible antes de entrar en el bucle (`for`) — es crucial en este curso. Los malos hábitos surgen del desconocimiento de estos hechos precisos. El orden correcto de las condiciones en `if`s encadenados o anidados, o la idoneidad de una estructura `switch`, solo puede dominarse mediante ejercicios. La práctica es la clave para aprender a programar; no hay sustituto.

Lo que sigue en este curso depende enteramente de dominar los condicionales y los bucles. Lo que viene a continuación tiene más que ver con la organización del código y, finalmente, con potentes estructuras de datos para modelar problemas; pero lo que la programación realmente es — razonar, ser capaz de ir del problema a la solución, el algoritmo — ya ha sido cubierto.

2·15.1 Punto de control

1) Cada uno de los siguientes fragmentos compila sin error, pero cada uno esconde uno de los errores clásicos mencionados anteriormente. Encuéntralo y describe lo que el código **realmente** hace.

A.

```
int n;
cin >> n;
if (n = 0) {
    cout << "n is zero" << endl;
}
```

B.

```
int i = 1;
while (i <= 10);
{
    cout << i << endl;
    ++i;
}
```

C.

```
// intended: print 1, 2, ..., 10
for (int i = 0; i <= 10; ++i) {
    cout << i << " ";
}
```

2) Para cada tarea, decide si encaja mejor un bucle `for` o `while`, y justifícalo en una línea:

   2.a) imprimir la tabla de multiplicar de un número `n`, desde `n·1` hasta `n·10`;

   2.b) seguir pidiendo una contraseña hasta que el usuario escriba la correcta;

   2.c) sumar los primeros `100` números naturales;

   2.d) leer números hasta que el usuario introduzca uno negativo y luego informar de cuántos fueron positivos.

3) Dado

```
unsigned int a = 3, b = 5;
cout << a - b << endl;
```

¿qué imprime esto y por qué? ¿Qué particularidad de los enteros sin signo ilustra?

4) La siguiente cadena pretende calificar un `score` de `0` a `10`, pero un `10` se informa como `"Aprobado"`. Explica por qué, y reordena las ramas para corregirlo.

```
if (score >= 5) {
    cout << "Aprobado" << endl;
} else if (score >= 7) {
    cout << "Notable" << endl;
} else if (score >= 9) {
    cout << "Sobresaliente" << endl;
}
```

5) Explica por qué comparar dos valores `double` provenientes de cálculos distintos con `==` es arriesgado, incluso cuando, sobre el papel, los dos resultados deberían ser idénticos.

Pasamos ahora a dos temas importantes: primero, la organización del código mediante subprogramas; segundo, la construcción de verdaderas estructuras de datos que dotarán a tus programas de las capacidades que requieren.

Referencias

BÖHM, Corrado y JACOPINI, Giuseppe, 1966. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. *Communications of the ACM*. 1966. Vol. 9, no. 5, p. 366-371. DOI **10.1145/355592.365646**.

El teorema que dio fundamento formal a la programación estructurada: cualquier función computable puede expresarse usando únicamente secuencia, selección e iteración, sin necesidad de saltos `goto` sin restricción. El original es denso y está cargado de notación, pero el resultado en sí es lo suficientemente sencillo como para enunciarse en una frase y constituye la justificación teórica de la prohibición de `goto`, `break` y `continue` en este curso.

DE MORGAN, Augustus, 1847. *Formal Logic: or, The Calculus of Inference, Necessary and Probable*. London: Taylor, Walton.

El texto fundacional de la lógica simbólica como cálculo, que introdujo lo que se formalizaría después como las leyes de De Morgan: las reglas de dualidad de la negación sobre la conjunción y la disyunción con las que todo programador se encuentra en la simplificación booleana. De interés principalmente histórico, pero merece citarse para arraigar el curso en la tradición intelectual que conecta matemáticas, lógica y computación.

ROSEN, Kenneth H., 2019. *Discrete Mathematics and Its Applications*. 8. New York: McGraw-Hill Education. ISBN 978-1-259-67651-2.

La referencia estándar de matemáticas discretas en los planes de estudio de informática. Cubre lógica, técnicas de demostración, conjuntos, relaciones, grafos, árboles y combinatoria desde una perspectiva informática constante. La 8.ª edición está bien estructurada tanto para el autoaprendizaje como para el apoyo a las clases. Lectura de fondo recomendada para los estudiantes que deseen comprender los fundamentos matemáticos del diseño de algoritmos y la corrección de programas.

Tarea 4: Estructuras de control. Bucles simples

Resuelve cada problema dentro de `main()` usando bucles y condicionales. No se permiten estructuras de datos — las veremos más adelante. Los problemas marcados con 🧊 son desafiantes; no dudes en pedir ayuda. Los problemas marcados con 🧊🧊 son solo para los estudiantes más decididos.

1 Escribe un programa que encuentre el valor más grande, el más pequeño, y la media aritmética de una serie de N enteros introducidos por teclado. Pide N primero, luego lee N valores en un bucle `for`. Se permiten valores repetidos. Redondea la media al entero más cercano con: › Sol

```
int av = float(sum) / N + 0.5;
```

Por ejemplo: `3 3 3 3` ⟶ mayor: 3, menor: 3, media: 3. Y: `-3 4 5 5 4 5 5 5` ⟶ menor: −3, mayor: 5, media: 3.

2 Escribe un programa para calcular el factorial $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$ de un entero `n`. Usa un bucle `while`. Prueba con `n = 6` (720) y `n = 12` (479001600). ¿Qué ocurre con `n > 10` usando `int`? › while › for

HINT: $n!$ cuenta el número de formas de ordenar n objetos distintos en fila. Es el producto de todos los enteros desde 2 hasta n inclusive.

3 Escribe un programa que encuentre todos los múltiplos de 35 en el rango $[1500, 2700]$ y los imprima separados por comas. › while › for › steps

HINT: Itera de `1500` a `2700` e imprime cada número divisible por 35. Alternativamente, avanza en el bucle con incrementos de 35.

```
1505, 1540, 1575, 1610, ..., 2660, 2695                    display
```

4 Escribe un programa que lea caracteres hasta que se introduzca un punto `'.'`, luego imprime el recuento de vocales, letras mayúsculas y caracteres no alfabéticos. Usa `cin.get()` para leer un carácter cada vez — al contrario que `cin >>`, no omite espacios, tabuladores ni saltos de línea: › Sol

```
char c;
c = cin.get();
while (c != '.') {
    // your code here
    c = cin.get();
}
```

Una forma compacta equivalente usa la asignación como condición del bucle:

```
char c;
while (c = cin.get() and c != '.')
    // your code here
```

Aquí la asignación `c = cin.get()` actúa tanto como sentencia como expresión — un idioma habitual en C/C++. Cuando quieras omitir los espacios en blanco, puedes usar `cin >> c` en su lugar.

5 Lee un entero no nulo `n` por teclado (valida que no sea `0`). Luego lee una secuencia de enteros terminada por `0` y determina si `n` aparece en

la secuencia. Deja de leer en cuanto se encuentre `n` o se introduzca `0`, e imprime si `n` fue introducido o no. › Sol

6 Lee una secuencia de caracteres terminada por un punto `'.'` y determina si la cadena `abc` aparece en ella. Deja de leer en cuanto se encuentre `abc` o se alcance el punto, e imprime si `abc` fue introducido o no. › Sol

7 Lee una secuencia de enteros terminada por `0` y encuentra las posiciones de la primera y última aparición de `12`. Si `12` no aparece, imprime `0` para ambas. Por ejemplo: › Sol

$$8 \quad 6 \quad 1 \quad 12 \quad -9 \quad 12 \quad 12 \quad 33 \quad 0$$

debería dar como salida `4` y `7`.

HINT: Usa las variables `int first, last, pos, num` para registrar la primera aparición, la última aparición, la posición actual y el número actual.

8 Lee una secuencia de números naturales terminada por `0`, o detente antes si la suma acumulada de los elementos anteriores es igual al número actual. Por ejemplo, `1 1 2 5 0`: la suma de `1+1` es igual al siguiente número `2`, así que para cuando se lee `2`. Pero `1 3 1 2 0` no tiene tal coincidencia y lee hasta `0`. › Sol

HINT: Usa `int num, sum`. Lee un número primero, luego itera mientras `num != 0` y `sum != num`; dentro del bucle, suma `num` a `sum` y lee el siguiente número.

9 El último dígito (en base `10`) de un entero es `n % 10`; eliminarlo significa dividir por `10`. Construye un programa que imprima los dígitos de un entero en orden inverso. Por ejemplo, `2021` imprime `1202`. › Sol

10 Dado un número natural n, determina si es un cuadrado perfecto, es decir, si existe un número natural m tal que $m^2 = n$.

HINT: Prueba cada entero `i` a partir de `1` con un bucle `while`. Para en cuanto `i*i >= n` y comprueba si `i*i == n`. › Sol

11 Construye un programa que imprima todos los divisores de un número dado. Por ejemplo:

```
12  ->  1 2 3 4 6 12                                    example
7   ->  1 7
```

› Sol

12 Dado n, imprime todos los números en $[1, n]$ que sean divisibles por 3 o por 5 (o por ambos). Por ejemplo, para $n = 20$:

```
3 5 6 9 10 12 15 18 20                                  example
```

› Sol

13 Dado un número n, determina si es primo. › Sol

HINT: Prueba cada entero `i` desde 2 mientras `i < n`. Si algún `i` divide `n`, para; luego comprueba si el bucle terminó porque se encontró un divisor o porque no existe ninguno.

14 Dado n (por ejemplo, $n = 10000$), calcula la suma $1 + 2 + 3 + \cdots + n$ usando un bucle, luego verifica que el resultado coincide con la fórmula cerrada $\frac{n(n+1)}{2}$. Imprime ambos valores. › Sol

15 Evalúa la siguiente serie para un número real x:

$$s = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Suma solo los primeros términos hasta `const int MAXPOW = 10`.

HINT: Calcula cada potencia a partir de la anterior (multiplicando por x) y cada factorial a partir del anterior (multiplicando por `i`), de modo que el cuerpo del bucle no necesite ni `pow()` ni un bucle factorial anidado. › Sol

16 Evalúa la misma serie que el ejercicio 15, pero detente cuando el término que se está sumando caiga por debajo de `const float EPSILON = 0.000001`. › Sol

17 Usando el programa del factorial del ejercicio 2, calcula coeficientes binomiales:

$$\binom{x}{y} = \frac{x!}{y!(x-y)!}$$

Pide al usuario `int x, y` y minimiza el número de multiplicaciones realizadas. › Sol

18 Lee un número natural n y calcula el n-ésimo término de la sucesión de Fibonacci (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...), definida por $f_0 = 0$, $f_1 = 1$, y $f_n = f_{n-1} + f_{n-2}$. La sucesión crece exponencialmente — ten cuidado con n grande. ($f_{92} = 7540113804746346429$) › Sol

19 `19.countdigits.cpp` Lee un entero e imprime cuántas veces aparece cada dígito (0–9). Ignora el signo si es negativo. › Sol

```
1234321  ->  1:1  2:2  3:2  4:1                    example
100      ->  0:2  1:1
-2023    ->  0:1  2:2  3:1
```

20 `20.collatz.cpp` Lee un entero positivo e imprime su sucesión de Collatz: si es par, divide por 2; si es impar, multiplica por 3 y suma 1. Para en 1. Imprime también el número de pasos. › Sol

```
6   ->  6 3 10 5 16 8 4 2 1  (8 steps)           example
27  ->  27 82 41 ...  1  (111 steps)
```

21 `21.numwords.cpp` Lee caracteres hasta un punto `'.'` y cuenta el número de palabras. Una palabra se define como una secuencia maximal

de letras `a–z`. Usa `cin.get()` — `cin >>` no puede leer espacios en blanco. › Sol

```
hello world.                  ->  2 words                    example
 . one two   three   four.    ->  4 words
nospaceshere.                 ->  1 word
```

Considera el seguimiento del estado antes y después de leer cada carácter. Comienza en el estado `inWord = false`. Cuando leas una letra (`a–z`) y no estuvieras antes dentro de una palabra, incrementa el contador de palabras y pasa al estado `inWord = true`. Si el carácter leído no es una letra, vuelve al estado `inWord = false`. (Máquinas de estados finitos)

22 🧊🧊 El siguiente juego se juega con n objetos. La máquina decide quién va primero; luego el usuario y la máquina se turnan, retirando cada uno 1, 2 o 3 objetos. El jugador que toma el último objeto **pierde**. Construye un programa en el que la máquina siempre gane. › alice-bob › machine

HINT: Determina qué posiciones son **perdedoras** para el jugador que mueve (todos los movimientos posibles llevan a una posición ganadora para el oponente):

- $n = 1$: debe tomar 1 $\longrightarrow$ pierde.
- $n = 2, 3, 4$: puede alcanzar $n = 1$ $\longrightarrow$ gana.
- $n = 5$: cualquier movimiento deja una posición ganadora para el oponente $\longrightarrow$ pierde.

El patrón se repite cada 4 pasos: las posiciones donde `n % 4 == 1` son posiciones perdedoras. La estrategia de la máquina:

1. Si `n % 4 == 1`, deja ir primero al usuario.
2. En caso contrario, ve primero y toma `(n-1) % 4` objetos, dejando una posición perdedora para el usuario.
3. En cada turno posterior, si el usuario toma k objetos, toma $4 - k$, restaurando siempre una posición `n % 4 == 1` para el usuario.

o O o

Tarea 5: Estructuras de control. Bucles dobles

En esta tarea practicamos bucles anidados (un bucle, `for` o `while`, dentro de otro). Al principio pueden resultar difíciles porque los límites del bucle interno suelen depender de la variable del bucle externo. En los ejercicios de figuras siempre usamos bucles `for`-`for`. No uses modismos de cadenas de C++ como multiplicar una cadena por un número.

Los tamaños de las figuras están parametrizados de modo que su tamaño depende de una variable (`size` u otro nombre); dale un valor de muestra y ejecuta el programa, o solicita ese valor al usuario si lo prefieres.

Las figuras pueden resolverse con o sin `if`-`else`; el enfoque preferido es evitar condicionales complejos y en su lugar repetir bloques de código simples, cada uno responsable de una parte distinta de la figura. Espera algo de prueba y error hasta que la figura quede bien.

Los problemas marcados con 🧊 son desafiantes; no dudes en pedir ayuda. Los problemas marcados con 🧊🧊 están diseñados exclusivamente para los estudiantes más decididos.

1 Imprime un cuadrado relleno. Ejemplo con `side = 4`: › Sol

```
****                                                    example
****
****
****
```

Declara `int side = 4` y comienza con una sola fila de asteriscos:

```
for (int i = 0; i < side; ++i)
    cout << '*';
cout << endl;
```

Observa que `endl` va **después** del bucle interno — imprimirlo dentro rompería cada fila en líneas separadas. Ahora envuelve ese bloque en un bucle externo para repetirlo `side` veces:

```
for (int i = 0; i < side; ++i) {
    for (int j = 0; j < side; ++j)
        cout << '*';
    cout << endl;
}
```

Ambos límites de bucle son iguales porque la figura es un cuadrado. ¿Podrías usar la misma variable de índice `i` para ambos bucles?

```
for (int i = 0; i < side; ++i) {
    for (int i = 0; i < side; ++i)  // ← mismo nombre i?
```

Completa el programa y compruébalo.

2 Imprime un cuadrado vacío con `side = 5`: › Sol

```
*****                                                   example
*   *
*   *
*   *
*****
```

3 Imprime un cuadrado vacío con borde doble, con `side = 6`: › Sol

```
******
******
**  **
**  **
******
******
```
example

4 Imprime un rombo relleno con `sideHalf = 4`: › Sol

```
   *
  ***
 *****
*******
 *****
  ***
   *
```
example

5 Imprime un rombo vacío con `sideHalf = 3`: › Sol

```
  *
 * *
*   *
 * *
  *
```
example

6 Imprime el siguiente triángulo espaciado con `side = 6`. Los puntos que se muestran a continuación **no** se imprimen — marcan espacios para ayudarte a contar: › Sol

```
.....*
....*.*
...*.*.*
..*.*.*.*
.*.*.*.*.*
*.*.*.*.*.*
```
example

7 Imprime una copa de vino con semiancho `widthHalf = 5`: › Sol

```
***********
 *********
  *******
   *****
    ***
     *
     *
     *
     *
     *
  *******
***********
```
example

8 Imprime el siguiente triángulo con `height = 3`: › Sol

```
1                                                                    example
12
123
1234
```

9 🧊 Imprime el siguiente triángulo para cualquier altura dada (ejemplo con `height = 6`): › Sol

```
1                                                                    example
23
456
7890
12345
678901
```

10 Construye un programa que, dado un número, calcule la suma de todos sus divisores propios (incluido el 1, excluido el propio número), luego compruebe si el número es igual a esa suma. Por ejemplo: › Sol

```
12  ->  1+2+3+4+6 = 16  ->  12 is not perfect                        example
28  ->  1+2+4+7+14 = 28  ->  28 is perfect
```

HINT: Un número con esta propiedad se llama número perfecto.

11 🧊 Imprime los primeros N números primos. Pide N al usuario o usa $N = 100$. › Sol

12 Construye una tabla de multiplicar usando bucles. Usa `<iomanip>` y `setw(4)` para alinear las columnas. Ejemplo de salida (los puntos representan espacios — imprime espacios reales en tu solución): › Sol

```
...1...2...3...4...5...6...7...8...9..10                             example
...2...4...6...8..10..12..14..16..18..20
...3...6...9..12..15..18..21..24..27..30
...4...8..12..16..20..24..28..32..36..40
...5..10..15..20..25..30..35..40..45..50
...6..12..18..24..30..36..42..48..54..60
...7..14..21..28..35..42..49..56..63..70
...8..16..24..32..40..48..56..64..72..80
...9..18..27..36..45..54..63..72..81..90
..10..20..30..40..50..60..70..80..90.100
```

Usa `const int FROM = 1` y `const int TO = 10`.

13 Construye un programa que, dado un número, calcule el número de dígitos decimales que contiene. Por ejemplo, `1492` tiene `4` dígitos. El número `0` debe imprimir `1` — trata este caso límite con cuidado. › Sol

14 Construye un programa que tome un número y una posición, e imprima el dígito en esa posición (contando desde la derecha, empezando en 1). Para `1492` y posición `2`, imprime `9`; para posición `0` imprime nada; para posición `5` imprime `0`. → Sol

15 🧊🧊🧊 Imprime una onda de sierra con `height = 5` y `nTeeth = 3`. › Sol

```
    *         *         *                                                 example
   * *       * *       * *
  *   *     *   *     *   *
 *     * *       * *       *
*       *         *
```

o O o

Part 3 Abstracción Procedimental

Contenido de esta parte Contenido del libro

3·1 Diseño Descendente 102
3·2 Procedimientos y Funciones. Parámetros 103
3·3 Recursión 119
▸ Tarea 6: Subprogramas 128
▸ Tarea 7: Subprogramas++ 132
▸ Tarea 8: Pre-estructuras 136

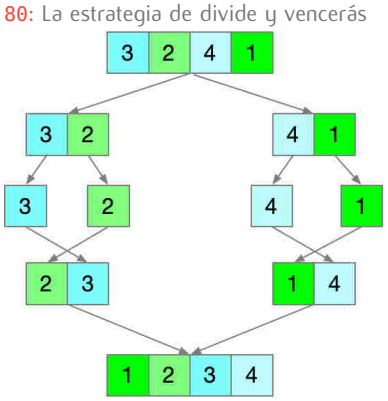
Abstracción de datos se refiere a modelar la información esencial contenida en los datos.
Abstracción procedimental consiste en modelar grupos de acciones organizándolas en subprogramas

Una de las obras más influyentes de Niklaus Wirth, autor de Pascal y Modula-2, lleva por título «Algorithms + Data Structures = Programs», publicada en 1976. En esencia, este libro refleja la importancia de organizar no solo las estructuras de datos, sino también las acciones en subprogramas.

3·1 Diseño Descendente

Cuando consideramos la solución a un problema, inicialmente lo imaginamos como un todo. Después, como vimos con Pólya (Sección 1·4), lo dividimos en partes más pequeñas y, por lo general, más fáciles de resolver. Esta técnica se denomina divide y vencerás. En realidad, este término se remonta al Imperio Romano, donde hacía referencia a la ventaja estratégica de mantener dividido al adversario para obtener poder sobre él. Sin embargo, sigue siendo válido en otros ámbitos. Cuando un problema supera nuestra capacidad para resolverlo en su totalidad, es mejor dividirlo en partes más pequeñas.

80: La estrategia de divide y vencerás

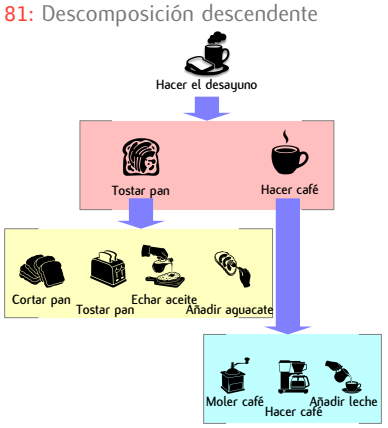


El mecanismo de divide y vencerás puede entenderse de dos maneras:

top-down se comienza comprendiendo el todo y, progresivamente, se profundiza en los detalles

bottom-up no se conoce exactamente cuál será la solución completa ni a qué se aplicará, pero se empieza combinando piezas pequeñas ya resueltas y agrupándolas en una solución.

81: Descomposición descendente



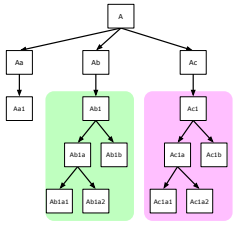
El primer enfoque es el más importante para nosotros, ya que los ingenieros se enfrentan habitualmente a problemas a gran escala y deben apoyarse en soluciones ya establecidas.

3·1.1 Jerarquía y acoplamiento

Al construir las piezas de una descomposición descendente, debemos tener cuidado de mantener la jerarquía **desacoplada**. Es decir, los submódulos —las partes que construimos— deben ser independientes entre sí.

Esto es fácil de verificar siguiendo las llamadas que realiza cada módulo: si, para resolver su parte del problema, un módulo solo necesita llamar a sus propios submódulos, y cada uno de estos, a su vez, solo llama a submódulos de su propia rama, tenemos una descomposición jerárquica limpia. Si, por el contrario, un submódulo necesita llamar a otros fuera de su propia rama para hacer su trabajo, la descomposición está **acoplada**.

82: Ejemplo de una descomposición jerárquica limpia. Las flechas representan las dependencias de llamada que tiene cada submódulo



Examine nuevamente Figura 81: cada pequeña tarea es autónoma; no necesita llamar a ninguna de sus hermanas para cumplir su función. Esta independencia es precisamente lo que mantiene el desacoplamiento de la jerarquía. Volveremos a este concepto de forma más rigurosa cuando comencemos a escribir nuestros propios subprogramas: la **cohesión** y el **acoplamiento** son dos de los principales criterios que utilizaremos para evaluar si una descomposición es correcta.

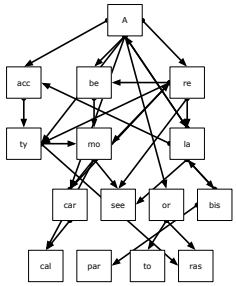
En resumen, la descomposición jerárquica ofrece tres ventajas clave:

más fácil de leer cada componente aborda un problema único y bien definido, lo que permite concentrarse en un aspecto a la vez. Este aislamiento hace que el código sea sencillo de seguir.

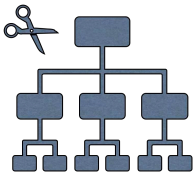
más fácil de modificar los cambios realizados en un componente no se propagan a otros. Es posible mejorar o refinar la implementación de un módulo concreto sin preocuparse por efectos secundarios involuntarios en otros lugares.

más fácil de reutilizar como las soluciones son independientes de su contexto y entorno circundante, pueden transferirse a otros dominios del problema con adaptación mínima o nula.

83: En una descomposición acoplada, los submódulos individuales quedan oscurecidos por una malla de llamadas interdependientes



84: La descomposición jerárquica facilita la lectura, modificación y reutilización de los programas



Como principio fundamental:

> Cada subprograma debe hacer una sola cosa

85: Cada subprograma ↔ un objetivo



Exploremos ahora qué nos ofrece C++ para dividir y conquistar nuestros programas.

3·1.2 Punto de control

1) ¿Cuál es la principal ventaja del diseño descendente?
2) ¿La escritura estructurada y ordenada de programas es un requisito del lenguaje de programación, o es responsabilidad del programador?
3) ¿Es siempre necesario escribir subprogramas?

3·2 Procedimientos y Funciones. Parámetros

Un subprograma es un conjunto de sentencias diseñadas para realizar una tarea concreta.

Consideremos el cálculo de una raíz cuadrada: lo único que necesita el cálculo es el número sobre el que operar. Así, construimos un subprograma para resolver exactamente ese problema: le damos el número del que queremos la raíz cuadrada y esperamos recibir otro número. Nada más, nada menos.

Cuando los estudiantes empiezan a programar, están acostumbrados a leer e imprimir valores directamente, por lo que es muy habitual ver un primer subprograma que pide al usuario un número e imprime la respuesta por sí mismo —por ejemplo, imprimiendo `"The square root of x is 2.0"` desde dentro del subprograma. Esto es exactamente lo contrario de para qué existen los subprogramas: un subprograma que solo sabe comunicarse con el teclado y la pantalla no puede reutilizarse en ningún otro lugar. Encontrarás abundantes soluciones en línea (y el código generado por LLMs copia fácilmente este hábito) que leen la entrada e imprimen el resultado dentro del propio cuerpo del subprograma; resiste esa tentación.

Si un subprograma no pudiera recibir información para trabajar, sería casi inútil. Le proporcionamos esa información a través de los **parámetros**. La idea es la misma en todos los lenguajes de programación: cuando escribes `sqrt(x)`, `sqrt` nombra el cálculo, mientras que `x` —que acompaña a la llamada entre paréntesis— es el parámetro, el valor sobre el que quieres que opere ese cálculo.

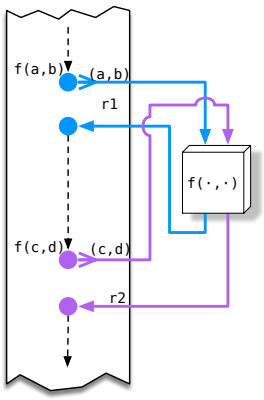
3·2.1 Ejemplo

Construir una función en C++ sigue esta sintaxis:

```C++
int fact(int n)
{
    int r = 1;
    for (int i = 2; i <= n; ++i) {
        r *= i;
    }
    return r;
}
```

Cuatro ingredientes componen esta definición, y puedes señalar cada uno directamente en el código anterior: un **nombre** (`fact`); una **lista de parámetros** entre paréntesis —aquí, un único entero que, dentro de la función, se llama `n`; un **cuerpo** —el ámbito delimitado por llaves, donde ocurre el trabajo real; y un **tipo de retorno** (`int`, escrito antes del nombre), acompañado de la sentencia `return` que devuelve el valor calculado, `r`, a quien llamó a la función.

86: Llamadas a un subprograma desde distintos puntos de otros subprogramas



3·2.2 Definición y Declaración de Procedimientos y Funciones. Parámetros Formales

Para construir un subprograma necesitamos los mismos cuatro ingredientes que antes: un **nombre**, una lista de **parámetros**, un **cuerpo** y un **tipo**. C++, sin embargo, tiene sus propias reglas sobre dónde pueden colocarse los subprogramas: todos deben estar al mismo nivel; el anidamiento no está permitido, es decir, no se puede declarar un subprograma dentro de otro. Ten en cuenta que, de hecho,

```
int main()
{
  // ..
```

```
}
```

es en sí mismo un tipo de subprograma: es lo que el sistema operativo llama para ejecutar el conjunto de acciones de nuestro programa. Así pues, desde el punto de vista del sistema operativo, `main()` también es un subprograma, razón por la cual no podemos escribir subprogramas dentro de `main()`; deben escribirse antes o después de él.

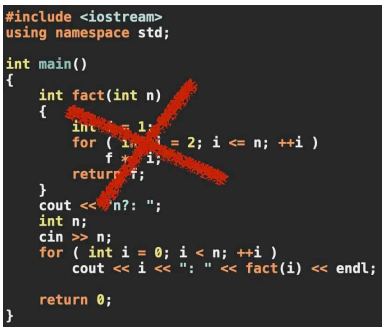
Por otro lado, en C++ todo lo que usemos —una variable, un tipo, un subprograma— debe haber sido declarado previamente antes de usarlo (o llamarlo). Por ello, hasta ahora nuestros subprogramas han tenido que escribirse antes de `main()`:

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

int fact(int n)
{
    int f = 1;
    for ( int i = 2; i <= n; ++i ) {
        f *= i;
    }
    return f;
}

int main()
{
    cout << "n?: ";
    int n;
    cin >> n;
    for ( int i = 0; i < n; ++i ) {
        cout << i << ": " << fact(i) << endl;
    }
    return 0;
}
```

87: El anidamiento no está permitido en C++



Pero esta tecnicidad impone una presentación incómoda del código, exactamente lo contrario de lo que hemos venido defendiendo: la descomposición descendente.

Nos obliga a escribir los detalles antes que la idea principal. Es como si un libro comenzara por sus apéndices: para descubrir de qué trata realmente el libro, habría que empezar a leer en la página 551, vadear detalles que no significan nada todavía, porque nadie nos ha contado antes la visión de conjunto.

3·2.2.1 Procedimientos frente a Funciones

Cuando llamas a un subprograma, normalmente esperas recibir alguna información de la operación invocada. Los subprogramas que devuelven información se denominan **funciones**, mientras que los subprogramas que no devuelven nada se denominan **procedimientos**. Las funciones son el tipo más común, y a veces nos referimos a todos ellos simplemente como funciones; sin embargo, la sintaxis y el uso difieren entre unos y otros.

```
void nameOfProcedure(param, param,...) {
  // code
}
```

```
type nameOfFunction(param, param,...) {
  // code
  return something_type
}
```

88: Las funciones devuelven un único valor del tipo que se declara que retornan. Los procedimientos actúan sobre entidades externas como la pantalla, sonidos o internet, pero no devuelven nada. Sin embargo, pueden modificar sus parámetros, como veremos

```
int f()
{
    actions;
    return val;
}

void g()
{
    actions;
}
```

Como los procedimientos no devuelven nada, al ser llamados no hay nada que asignar a una variable. Las llamadas a procedimientos aparecen siempre al comienzo de una línea, sin signo `=` delante:

```
{
  ...
  printLines(3);
  ...
}
```

Las funciones, por el contrario, siempre devuelven un valor, por lo que en aproximadamente el 99% de los casos tendrán un signo `=`, o formarán parte de una expresión que usa el valor devuelto, o quizás sirvan como parámetro en la llamada a otro subprograma.

```
{
  ...
  int r = fact(n);
  float term = x / fact(++n);
  printLines(fact(2));
  ...
}
```

¿Puedes imaginar algún uso de un procedimiento sin parámetros? No están prohibidos en absoluto, y el único propósito práctico que podrían tener es modificar algo fuera de nuestro programa. Por ejemplo, limpiar la pantalla:

```
  ...
  clrscr();
  ...
```

3·2.2.2 Punto de control

1) ¿Es sintácticamente permisible escribir:

```
int x = f(g());
```

2) Si `p()` es un procedimiento y `f()` es una función, ¿es permisible la siguiente sintaxis?

   2.a) `int x = f()`

   2.b) `p(f());`

   2.c) `f(p());`

3) ¿Cuándo sería de interés práctico un procedimiento sin parámetros?

3·2.2.3 Interfaces. Prototipos

En algunos lenguajes, a pesar de ofrecer numerosos mecanismos para organizar algoritmos, subprogramas, métodos, etc., no proporcionan formas directas de escribir subprogramas en el orden deseado. El orden en que debe escribirse el código viene determinado por el funcionamiento interno del compilador.

C++ ha proporcionado una solución a este problema desde sus inicios. La comprobación de argumentos de función —lo que hoy llamamos **prototipo**— fue una de las primeras características que Stroustrup añadió a «C with Classes», y la hizo obligatoria ya en 1982, años antes de que el

89: Un prototipo informa al compilador de que existe un subprograma con esa interfaz y permite que las líneas de código posteriores lo utilicen

```
int f(int n);
```

90: Los prototipos no necesitan mostrar los nombres de los parámetros, salvo cuando un nombre facilita la lectura, por ejemplo, para parámetros con valores por defecto

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

lenguaje se rebautizara como C++ (1983) o alcanzara su primer estándar, C++98[△]. Un prototipo nos permite ordenar nuestros subprogramas como queramos: el único requisito es declarar por adelantado que existirá un subprograma con un nombre y unos parámetros dados; su definición real, el cuerpo, puede aplazarse hasta más adelante:

△ B. Stroustrup, *A History of C++: 1979–1991*.

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

int main()
{
    int fact(int n);   // prototype: fact() exists,
                       // its definition comes later

    cout << "n?: ";
    int n;
    cin >> n;
    for ( int i = 0; i < n; ++i )
        cout << i << ": " << fact(i) << endl;

    return 0;
}

int fact(int n)
{
    int f = 1;
    for ( int i = 2; i <= n; ++i )
        f *= i;
    return f;
}
```

De este modo, comenzamos a leer el programa de forma natural: nos encontramos con `main()` —el esquema de lo que hace el programa— de inmediato, y solo buscamos los detalles de `fact` después, si nos interesan.

Prototipos o **interfaces**
relajan el orden de colocación de los subprogramas facilitando su lectura natural desde la idea abstracta que queremos resolver hasta los detalles de implementación, de arriba hacia abajo.

3·2.2.4 Organización Descendente del Programa

Un archivo fuente C++ debe seguir este mismo orden descendente a lo largo de todo él, avanzando desde los elementos más generales hasta los más específicos:

1. Inclusiones (`#include`), ordenadas de cabeceras más generales a más específicas.
2. Constantes (`const`).
3. Definiciones de tipos (`typedef`).
4. `int main()`, acompañado de los prototipos de los subprogramas que necesite.
5. Cuerpos de los subprogramas, ordenados del más abstracto al más detallado.

Hemos visto la práctica habitual de incluir cabeceras al comienzo del `.cpp`, lo cual tiene sentido, ya que documenta las dependencias externas del programa. Cabe señalar aquí que la inclusión de cabeceras implica definiciones, y cuando se incluyen muchas cabeceras, algunas definiciones

pueden entrar en conflicto inadvertidamente con otras de otras cabeceras, por lo que una regla saludable de inclusión, en general, es incluir primero las cabeceras más abstractas y las propias del programa al final.

Cuando escribimos `#include <cmath>` le pedimos al preprocesador que busque el archivo `cmath` en los directorios estándar que nuestra configuración de C++ tiene establecidos. En un programa complejo formado por más de un archivo `.cpp`, es necesario disponer de definiciones comunes para todos los archivos, y esas definiciones (constantes, tipos, prototipos) se recogen en cabeceras `.h` como `geometry.h` que nuestros archivos `.cpp` incluirán: `#include "geometry.h"` sin ningún `<>`, indicando directamente al preprocesador que localice el archivo en la ruta que escribimos, relativa a nuestro propio programa.

Así pues, para la inclusión de cabeceras tiene sentido un orden de lo general a lo particular.

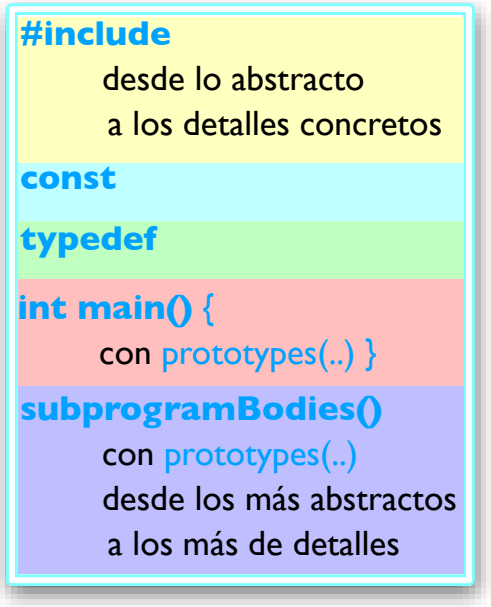
Las constantes, por otro lado, son los elementos más seguros de nuestro lenguaje, pero muchos otros pueden depender de ellas, por lo que su ubicación debe estar justo después de la inclusión de cabeceras, también ordenadas de lo general a lo específico (la constante π, por ejemplo, difícilmente dependería de ninguna otra).

Los tipos, por su parte, suelen depender de las constantes (los arrays especialmente) y deben ir justo después de ellas, ya que definen la organización general de nuestros datos, no son perjudiciales, no demandan ni memoria ni tiempo de nuestro programa final, y son necesarios para los subprogramas que siguen.

El `main()` es el índice, la guía de lo que nuestro programa pretende hacer, por lo que debe ser la primera «función» declarada, y en su parte superior, antes de cualquier declaración de variables, van los prototipos de los subprogramas que necesitará para realizar su trabajo —como declaración de dependencias, nada más de lo estrictamente necesario.

Después, finalmente, tras el `main()` vienen los cuerpos de los subprogramas, del más abstracto al más detallado.

Organización de arriba-abajo (Top-Down)



3·2.2.5 Punto de control

Diseña las interfaces para los siguientes subprogramas que calculan:

1) el área de un círculo
2) el máximo de tres números
3) el código ASCII de una letra
4) la letra que ocupa una posición dada en el alfabeto
5) el número de días entre dos fechas dadas (solo la interfaz)
6) si un número es par o impar
7) si una fecha dada cae en un año bisiesto

3·2.3 Llamada a Procedimientos y Funciones. Parámetros Reales

Ahora tenemos subprogramas con parámetros. En torno a ellos surgen cuatro preguntas una y otra vez, por lo que conviene fijar el vocabulario de una vez por todas:

parámetro formal el tipo y el nombre que damos a un parámetro en la declaración propia del subprograma. Esto se parece mucho al contador de un bucle `for`: `i` se declara en la cabecera del bucle, entre paréntesis, y aun así se comporta como si perteneciera al interior del cuerpo —es utilizable a lo largo de él y desaparece cuando el bucle termina (Figura 73). Un parámetro formal funciona igual: se escribe entre los paréntesis del subprograma, justo junto a su nombre, pero su ámbito se reduce al cuerpo exactamente como si se hubiera declarado dentro de él.

parámetro real el valor, o expresión, que suministramos en cada llamada individual al subprograma. La única regla es que debe ser asignable a su parámetro formal correspondiente, la misma regla de compatibi-

lidad que rige cualquier asignación ordinaria. Los parámetros reales se emparejan con los formales uno a uno, en el orden en que están escritos, no por nombre.

valor de retorno cuando un subprograma devuelve un valor (los procedimientos, como veremos en breve, no devuelven nada en absoluto), ese valor debe ser compatible en asignación con el propio tipo del subprograma, y cada sentencia `return` devuelve exactamente un valor. C++ permite varias sentencias `return` distribuidas por una función, pero por ahora nos ceñiremos a la convención de escribir una sola, al final del cuerpo, una vez que el valor a devolver se haya calculado completamente —un único punto de salida es mucho más fácil de leer y de razonar.

valores por defecto los parámetros formales al final de la lista de parámetros pueden recibir un valor por defecto. Esto evita repetir, en cada llamada, el valor que ese parámetro toma habitualmente: si una llamada lo omite, se utiliza el valor por defecto escrito en la declaración. C++ solo permite esto para un sufijo final de la lista —una vez que un parámetro tiene valor por defecto, todos los parámetros posteriores también deben tenerlo, o el compilador ya no podría determinar qué valores de una llamada más corta corresponden a qué parámetros.

El fragmento siguiente no define `combinatorial`, `printComplex`, `printlns` ni `exseries` —solo declara sus prototipos, para mostrar, una al lado de la otra, varias formas de llamarlos:

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

int main()
{
    int combinatorial(int n, int p);
    void printComplex(float real, float im);
    void printlns(int n = 1);
    float exseries(float x, float precis);

    int a = 5;
    float p_im = -1.5, x = 3.0, y = -2.0, r;

    cout << combinatorial(a, 2);
    printComplex(2 + 3.0, p_im);
    printComplex(x, y);
    printlns();
    printlns(3);
    r = exseries(0.5, 0.00001);

    return 0;
}
```

Obsérvese `printComplex(2 + 3.0, p_im)`: la expresión `2 + 3.0` es de tipo `double`, pero `real` espera un `float` —se convierte implícitamente, exactamente como prometió la regla de compatibilidad de asignación para parámetros reales. Obsérvese también `printlns()` y `printlns(3)`: el mismo subprograma llamado con y sin su parámetro (con valor por defecto).

En los prototipos no es obligatorio indicar los nombres de los parámetros formales (salvo para los parámetros que tengan un valor por defecto,

que necesitan un nombre al que asociar dicho valor por defecto); puedes escribir solo lo que importa para la llamada en sí: el tipo.

```C++
int main()
{
    int combinatorial(int, int);
    void printComplex(float, float);
    void printlns(int n = 1);
    float exseries(float, float);
    // ..
}
```

3·2.4 Paso de Parámetros por Valor y por Referencia

Hemos aprendido que un subprograma puede devolver un valor, pero solo uno, una única pieza de información.

Sin embargo, es bastante habitual que un programa necesite un subprograma que devuelva dos o más valores, y eso simplemente no es posible con una función tal como la hemos definido hasta ahora.

Imaginemos que necesitamos una función para calcular las soluciones de una ecuación de segundo grado. Puede haber dos soluciones, por lo que podríamos sentirnos tentados a escribir algo como:

```C++
float, float quadEq(float a, float b, float c)
{
  return x1, x2;
}
```

Esto no compila: una función tiene exactamente un tipo de retorno y devuelve exactamente un valor, nunca una lista separada por comas de ninguno de los dos.

Para sortear esta limitación, C++ ofrece otro mecanismo.

Recuerda que los parámetros reales, hasta ahora, siempre se **copian** en los parámetros formales: ese es el mecanismo por defecto en la mayoría de los lenguajes de programación. Pero hay otra forma de pasar un valor a un parámetro formal: pasarlo **por referencia**.

Hasta ahora, cuando los valores se pasan por copia, lo que recibe el parámetro formal —y por tanto el subprograma— es una copia simple: exactamente igual que cuando una variable se asigna a otra, la copia no mantiene ninguna conexión posterior con el original. Puedes cambiar tu copia dentro del subprograma, pero el parámetro real original está a salvo; nada de lo que el subprograma haga con su propia copia puede alcanzarlo.

Pero C++ ofrece un mecanismo diferente para pasar parámetros, el paso del parámetro **por referencia**.

El C original solo podía pasar parámetros por copia o, para un acceso similar al de los punteros al original, pasando explícitamente un puntero a él. C++ añade una tercera forma, más controlada: el paso por referencia.

El paso de un parámetro por referencia utiliza esta sintaxis:

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

int main()
{
    void swap(int& a, int& b);

```

```
8       int var1 = 1;
9       int var2 = 2;
10      swap(var1, var2);
11      cout << var1 << ", " << var2 << endl;
12      return 0;
13  }
14
15  void swap(int& a, int& b)
16  {
17      int t = a;
18      a = b;
19      b = t;
20  }
```

En el ejemplo anterior, los parámetros se pasan por referencia. El procedimiento `swap` recibe referencias a las variables reales `var1` y `var2` cuando se llama en la línea 10. Una referencia, una vez modificada, modifica la propia variable a la que hace referencia; así, dentro de `swap`, la línea 18 modifica inmediatamente `var1` (a través de `a`), y la línea 19 modifica inmediatamente `var2` (a través de `b`). Cualquier otra llamada, desde `main()` o desde cualquier otro ámbito, enviaría las referencias correspondientes a `swap()`, y las variables correspondientes se verían afectadas exactamente de la misma manera.

De hecho, la declaración de referencias en C++ no se limita a los parámetros, pero en la práctica los parámetros son prácticamente el único lugar donde realmente se usan.

Recuerda: para recibir un parámetro real por referencia, declara el parámetro formal correspondiente como

```
type& name
```

El `&` puede ir junto al tipo (`type& name`), junto al nombre de la variable (`type &name`), o por sí solo con espacios a ambos lados (`type & name`) —a C++ no le importa el espaciado, pero el propio `&` debe estar presente.

Un error clásico: en una declaración de múltiples variables como `int& a, b;`, el `&` solo se une a `a` —`b` se declara como un `int` simple, no como una referencia. En caso de duda, declara una referencia por línea.

Como un parámetro de referencia da al subprograma acceso directo a la propia variable del llamador, solo una variable real —algo con una dirección, un *lvalue*— puede pasarse como argumento. Un literal o el resultado de una expresión, como ~~`swap(1, 2)`~~ o ~~`swap(a + 1, b)`~~, no tiene lugar al que apuntar la referencia, por lo que C++ se niega a compilarlo.

El ejemplo de intercambio anterior no podría escribirse en C++ de ninguna otra forma, salvo recurriendo a una manera mucho más torpe de envolver la información, o a punteros al estilo C.

El paso por referencia no solo es útil para devolver múltiples valores o para que un subprograma pueda modificar las variables del llamador; también evita el coste de copiar, que se vuelve significativo en cuanto las variables dejan de ser números individuales y pasan a ser estructuras más grandes como arrays y registros, que trataremos en el próximo capítulo. Sin embargo, en este curso no abordaremos este tipo de optimización de rendimiento sutil.

Aunque C++ permite pasar parámetros por referencia tanto en funciones como en procedimientos, muchos autores —e incluso lenguajes de programación enteros— prohíben el paso por referencia en las funciones, reservándolo para los procedimientos: en esa visión, una función solo debe comunicarse a través del único valor que devuelve, nunca modificando implícitamente sus parámetros.

Recuerda:

coincidencia exacta de tipos el paso por referencia requiere que los parámetros real y formal sean exactamente del mismo tipo.

efectos laterales las variables reales pasadas por referencia se modifican implícitamente, lo que hace el programa menos seguro. Es responsabilidad del programador hacer evidente que los cambios o los valores iniciales se establecen implícitamente en la llamada al subprograma.

Como ejemplo final, consideremos el subprograma de la ecuación de segundo grado que discutimos antes. Para ello, pasamos a, b y c de la ecuación $ax^2 + bx + c$ al procedimiento y esperamos que rellene los dos parámetros por referencia típicamente colocados al final de la lista de parámetros:

```
void quadEq(float a, float b, float c, float& x1, float& x2)
{
    ...
}
```

¿Qué pasa si quisieras determinar la naturaleza de la solución? Hay varias posibilidades que podríamos enumerar de la siguiente manera:

```
enum TYPEEQ {ANYX, ABSURD, LINEAR, DOUBLE, IMAG, TWODIFSOLS};
```

¿Cómo podrías modificar el procedimiento (o función) para recibir también esta información?

En resumen, el paso de parámetros por referencia conlleva las siguientes implicaciones:

- 👎 Susceptibilidad a problemas de efectos laterales
- 👎 Los parámetros reales deben ser variables del mismo tipo, lo que impide el uso de constantes o expresiones como parámetros reales, limitando así su aplicabilidad
- 👍 Mejor rendimiento, especialmente para estructuras de datos grandes, ya que no se requiere copia
- 👍 Capacidad de devolver dos o más valores

Por tanto,

> Usa `&` solo cuando necesites devolver más de un valor

3·2.5 Punto de control

1) ¿Cuántos valores de retorno puede tener una función?
2) Aparte del requisito de obtener dos o más resultados, ¿cuándo serían útiles los parámetros pasados por referencia?
3) ¿Puede una función recibir y modificar un parámetro pasado por referencia? ¿Por qué es apropiado o inapropiado este enfoque?
4) ¿Qué hace menos conveniente el envío de parámetros por referencia?
5) ¿Es necesario que los parámetros formales del prototipo tengan nombres?
6) ¿Qué es un valor por defecto? ¿Dónde deben colocarse en la lista de parámetros los parámetros con valores por defecto?

3·2.6 Criterios de Modularización

¿Cuándo usar subprogramas?

Hasta ahora hemos escrito muchos programas sin necesitar nunca dividir parte de ellos en un subprograma. Pero a medida que los programas crecen, el número de variables y pequeñas tareas que `main()` debe gestionar convierte un `main()` monolítico en algo difícil de seguir. Por otro lado, muchos programas necesitan realizar la misma operación, o extraer el mismo tipo de información de distintos valores, una y otra vez. Repetir el mismo cálculo una y otra vez no solo es tedioso, sino también difícil de mantener.

Tres criterios, en particular, ayudan a decidir cuándo un fragmento de código merece convertirse en un subprograma propio:

tareas diferentes cuando parte de un programa está dedicada a algo claramente distinto del objetivo del código circundante, extráela a un subprograma. Para hacerlo, necesitas saber exactamente de qué información depende ese objetivo separado —y esto es en sí mismo una buena prueba: si puedes nombrar el puñado de parámetros de los que depende un fragmento de código, y nada más, ese fragmento es naturalmente independiente del resto y un buen candidato a subprograma.

repetición el criterio anterior es con frecuencia, por sí solo, razón suficiente para producir módulos claros. Pero a veces un programa no es lo suficientemente complejo como para tener partes obviamente separables, y sin embargo necesita repetir la misma operación varias veces. Siempre que puedas precisar exactamente qué información necesita una operación y qué valor produce, esa repetición por sí sola es razón suficiente para convertirla en un subprograma, incluso cuando el resto del programa no tenga otras costuras naturales.

legibilidad incluso las partes pequeñas de un programa —las que podrían resolverse en una o dos líneas de código— merecen convertirse en un subprograma independiente si son importantes y desempeñan un papel bien definido. Toma la consulta «¿es esta una

letra mayúscula?», `'A' <= c and c < 'Z'`: podría expresarse con mucha más claridad como:

```bool isUpper(char c)
{
  return 'A' <= c and c < 'Z';
}
```

al final de nuestro archivo `.cpp`. Esta función podría modificarse fácilmente más adelante —para manejar letras acentuadas, por ejemplo— sin tocar el resto del algoritmo que la utiliza.

El siguiente programa lleva esto incluso más lejos: cada detalle vive al final, y leerlo de arriba a abajo cuenta toda la historia.

```// lcm.cpp
// juanfc 2026-09-09
// least common multiple

#include <iostream>
using namespace std;


int main()
{
    int lcm(int a, int b);

    cout << lcm(4, 6) << endl;
    cout << lcm(21, 6) << endl;
    return 0;
}

int lcm(int a, int b)
{
    int gcd(int a, int b);

    return (a*b)/gcd(a,b);
}

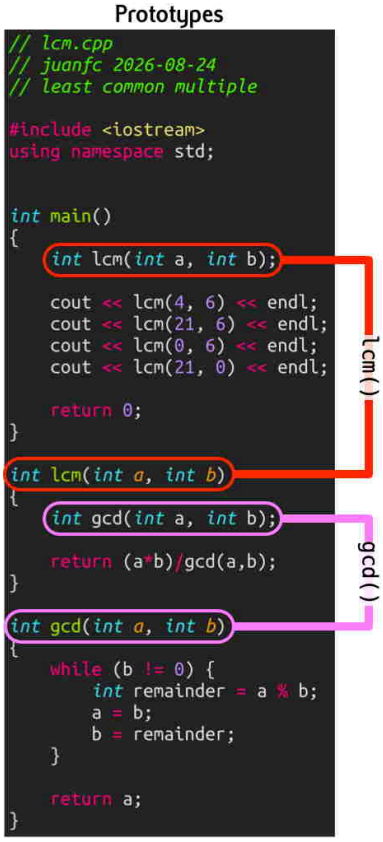
int gcd(int a, int b)
{
    int smallest(int a, int b);

    int m = smallest(a, b);
    while (a % m != 0 or b % m != 0)
        --m;
    return m;
}

int smallest(int a, int b)
{
    return (a < b) ? a : b;
}
```

91: Desde el índice que representa `main()` hasta los detalles, cada subprograma es independiente y puede colocarse en cualquier región de nuestro archivo `.cpp`. La declaración interna de prototipos al comienzo de cada subprograma documenta y posibilita su libertad para ordenarse según nuestra elección.

Prototypes



3·2.7 Variables Locales y Globales. Efectos Laterales

Cada variable que hemos declarado hasta ahora —dentro de `main()`, dentro del cuerpo de un subprograma o como uno de sus parámetros formales— es una **variable local**: existe solo dentro del ámbito del bloque que la declara, y desaparece en el momento en que ese bloque termina, exactamente como vimos antes en este capítulo (Figura 73).

No podemos continuar sin prohibirte el uso de lo que se denominan **variables globales**. Hasta ahora no hemos necesitado ninguna, por lo que debería quedar claro que podemos prescindir de ellas; sin embargo, plantean una tentación atractiva: simplemente declara la variable fuera del

ámbito de cualquier subprograma, y ya está, accesible desde cualquier lugar.

Las variables globales tienen una ventaja real: te ahorran pasar los mismos parámetros en todas partes. En un programa construido en torno a una pieza clave de información sobre la que casi todo trabaja, puede parecer extraño no simplemente declararla globalmente, una vez, antes de `main()`, como una gloriosa pieza de información compartida.

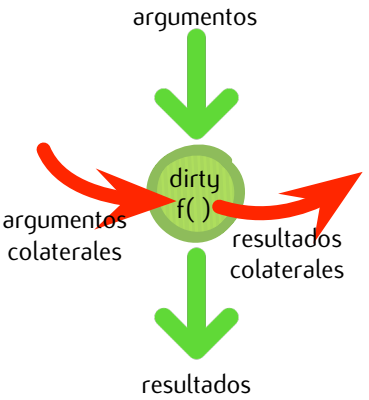
La mala noticia es que están completamente prohibidas en este curso.

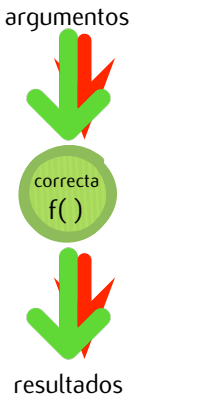
Los lenguajes más antiguos, como el Fortran o el BASIC primitivos, dependían en gran medida de las variables globales; eran otros tiempos. Los lenguajes modernos facilitan mucho, y hacen mucho más seguro, pasar información de forma explícita.

Las variables globales son muy difíciles de controlar. Viven en el ámbito global, por lo que cualquier ámbito, por profundamente anidado que esté, puede leerlas —y, mucho más peligrosamente, también puede escribir en ellas. Imagina un programa lleno de subprogramas que se llaman entre sí, cada uno modificando silenciosamente la misma variable global. Llamas a un subprograma para que haga una cosa específica y, tras el retorno de la llamada, esa variable importante —nunca parte de la lista de parámetros, nunca parte de la interfaz declarada del subprograma— ha cambiado de todos modos. Incorrecto.

Esto es exactamente lo que se denomina un **efecto lateral**: un cambio que un subprograma realiza sobre el estado del programa más allá del valor que devuelve, invisible a menos que vayas a leer el cuerpo completo del subprograma. Un subprograma que solo lee y escribe sus propios parámetros y variables locales no tiene efectos laterales; uno que alcanza y modifica una variable global siempre los tiene —y eso es precisamente lo que hace que las variables globales sean tan traicioneras para depurar.

92: La interfaz o prototipo de los subprogramas debe definir sus requisitos y el alcance de su operación. Si requieren variables externas o globales, quedan acoplados al entorno y no pueden reubicarse. Si operan sobre variables externas, esto es aún más problemático, ya que produce efectos laterales.





3·2.8 Punto de control

1) ¿Por qué los parámetros pasados por referencia no se consideran un problema de efectos laterales?
2) ¿Por qué muchos programas mal estructurados tienden a usar variables globales?

3·2.9 Precondiciones y Postcondiciones. Manejo de Situaciones Excepcionales

Como vimos en Sección 2·14, el tratamiento completo de las excepciones no es una preocupación central de un curso introductorio de programación. Aun así, dos ideas de ese título merecen fijarse adecuadamente ahora que tenemos subprogramas:

precondición lo que un subprograma espera que sea verdad —sobre sus parámetros, típicamente— antes de comenzar a hacer su trabajo. Verificar que se cumple una precondición es, por convención, responsabilidad del **llamador**; un subprograma solo promete comportarse correctamente cuando sus precondiciones se satisfacen.

postcondición lo que un subprograma garantiza que será verdad una vez que retorna —sobre su resultado y sobre cualquier otra cosa que tenga permitido cambiar— siempre que se haya cumplido su precondición. Esta es la promesa que el subprograma hace a quien lo llamó.

Tomemos `gcd(a, b)` de antes en este capítulo (Figura 91): tiene una precondición implícita —`a` y `b` deben ser ambos distintos de `0`— y una postcondición: el valor que devuelve es el máximo común divisor de `a` y `b`. Nada en el código lo decía explícitamente; el compilador no te habría impedido llamar a `gcd(0, 0)`, pero el **contrato** se habría roto de igual manera.

Si una situación dada se trata como una **excepción** —en lugar de como otra rama de la lógica ordinaria— depende de tres cosas:

cómo reaccionan el sistema operativo y el hardware cuando les pedimos que realicen una acción, distintos sistemas pueden reaccionar de forma diferente a la misma petición —la división por `0` siendo un ejemplo clásico.

lo que nosotros, los programadores, decidimos que cuenta como excepcional frente a una situación que el programa debe simplemente absorber como parte de su flujo normal. Es exactamente este nivel de tolerancia lo que convierte una situación en una excepción, y es exactamente para lo que sirve una precondición: declarar, para cada subprograma —y para el programa en su conjunto— qué espera y qué no espera recibir.

el tipo de aplicación que estamos construyendo supone una diferencia real si esto es un ejercicio de algoritmos simple, un programa comercial destinado a funcionar sin ser tocado durante años, o firmware embebido en un chip, donde reaccionar correctamente ante cada situación inesperada puede ser crítico.

Dicho todo esto, recuerda que en Sección 2·14 ya conocimos dos de los tres comandos que C++ usa para manejar excepciones —`try` y `catch`— y prometimos volver al tercero una vez que existieran los subprogramas. Ese tercer comando es `throw`: es la forma en que un subprograma informa de que se ha incumplido una de sus precondiciones, pasando un valor —a menudo solo un mensaje descriptivo— al `catch` que esté preparado para gestionarlo.

Las excepciones tienen mucho más sentido una vez que existen los subprogramas, y muy poco sin ellos: cuando un subprograma llama a otro, que a su vez llama a otro, una excepción lanzada en lo más profundo puede ser capturada varios ámbitos más arriba, sin que cada subprograma intermedio tenga que comprobarla y reenviarla manualmente, y sin que todo el programa tenga que bloquearse.

Un ejemplo sencillo, reutilizando `lcm`, `gcd` y `smallest` de antes en este capítulo (Figura 91), ahora protegido contra la división por cero que un `lcm(0, n)` sin verificar causaría:

```cpp
#include <iostream>
#include <stdexcept>
```

```
3  using namespace std;
4
5  int main()
6  {
7      int lcm(int a, int b);
8
9      try {
10         cout << lcm(4, 6) << endl;
11         cout << lcm(21, 6) << endl;
12         cout << lcm(0, 6) << endl;   // breaks the precondition
13     } catch (const domain_error& e) {
14         cout << "ERROR: " << e.what() << endl;
15     }
16     return 0;
17 }
18
19 int lcm(int a, int b)
20 {
21     int gcd(int a, int b);
22
23     return (a*b)/gcd(a,b);
24 }
25
26 int gcd(int a, int b)
27 {
28     if (a == 0 or b == 0)
29         throw domain_error("gcd requires a and b to be non-zero");
30
31     int smallest(int a, int b);
32
33     int m = smallest(a, b);
34     while (a % m != 0 or b % m != 0)
35         --m;
36     return m;
37 }
38
39 int smallest(int a, int b)
40 {
41     return (a < b) ? a : b;
42 }
```

Obsérvese también que `gcd` comprueba su precondición explícitamente, con un simple `if`, antes de calcular nada en absoluto♡. La llamada a `lcm(0, 6)` en la línea 12 rompe esa precondición dos llamadas más abajo —dentro de `gcd`, que lanza en la línea 29— y sin embargo `main()` la captura directamente, en la línea 13, sin que `lcm` necesite saber nada al respecto. Las líneas 10 y 11 ya se ejecutaron e imprimieron sus resultados; el resto del bloque `try`, todo lo que va después de la línea 12, simplemente se omite —exactamente el salto descrito anteriormente en este capítulo.

♡ Esto importa: una división real por cero —escribir `a % 0` y dejar que C++ la evalúe — no es algo de lo que un `try`/`catch` pueda rescatarte. Es comportamiento indefinido y en la práctica bloquea el programa directamente, exactamente como el `n/d` sin protección de Sección 2·14. Las excepciones solo ayudan con los errores que detectamos **antes** de que ocurran y que informamos **deliberadamente**, con `throw`.

Obsérvese que cuando decimos

```
throw domain_error("gcd requires a and b to be non-zero");
```

no estamos **devolviendo** un valor: esta vez `gcd` no le da a `main()` ningún `int` de vuelta. En cambio, abandona su propio trabajo —y el de `lcm`, que aún lo espera— y el objeto lanzado viaja por la pila de llamadas hasta que encuentra un `catch` cuyo tipo coincide. Esa coincidencia es el punto central: nuestro `catch (const domain_error&)` lo recibe porque un `domain_error` es exactamente lo que se lanzó y, al ser una excepción estándar, lleva el mensaje `.what()` que imprimimos. Un `string` simple no habría coincidido —razón por la cual lanzamos los tipos estándar de `<stdexcept>` en lugar de improvisar con un valor crudo.

3·2.10 Subprogramas inline en C++

Al llamar a un subprograma, es necesario generar e insertar código preparatorio antes de la llamada y código de reanudación después de ella para restaurar el ámbito actual. Igual que `const type&` es una optimización de C++ que evita la sobrecarga interna que conlleva pasar objetos, el compilador también proporciona una optimización disponible desde la versión 2.0 conocida como `inline`. Puedes anteponer la palabra clave `inline` a cualquier declaración de subprograma, y el compilador optimizará entonces el código generado eliminando la sobrecarga del cambio de contexto. En lugar de realizar una llamada a función real, el compilador sustituye el cuerpo de la función (típicamente pequeño) directamente en cada punto de llamada. Por ejemplo, cada invocación de `inline int fact(int n);` es reemplazada por el propio código del cuerpo.

En este curso no estamos particularmente centrados en las optimizaciones específicas del compilador, y por esta razón no encontrarás la optimización `inline` en la práctica, del mismo modo que no verás `const type&` en ningún paso de parámetros. Sin embargo, vale la pena entender por qué existen estas optimizaciones y quién puede beneficiarse de su uso.

3·2.11 Punto de control

1) ¿En qué nivel de la jerarquía de subprogramas es apropiado usar un mecanismo `try`/`catch`/`throw`?
2) ¿Es `throw` equivalente a la sentencia `return`?

3·3 Recursión

Un subprograma no está restringido a llamar a otros subprogramas: puede llamarse a sí mismo. Cuando lo hace, lo llamamos **recursivo**. Suena casi como hacer trampa —resolver un problema llamando al mismo subprograma que está intentando resolverlo— pero es una de las ideas más antiguas y elegantes de la programación, y C++ no necesita ninguna sintaxis especial para ello: una función puede aparecer dentro de su propio cuerpo exactamente igual que llamaría a cualquier otra función.

> Para entender la recursión, primero debes entender la recursión.

Ese chiste contiene toda la advertencia que existe en esta sección: la recursión, hecha sin cuidado, no lleva a ningún lugar salvo a dar vueltas en círculos. Hecha correctamente, convierte los problemas definidos naturalmente en términos de versiones más pequeñas de sí mismos en código que es casi una transcripción directa de su definición.

```C++
int main()
{
  void a();

  a();  // we do not want `main` to be recursive

  return 0;
}

void a()
{
```

```
  if (cond) {
    // BASE CASE, no a()
  } else {
    a();
  }
}
```

3·3.1 Concepto de Recursión

Un subprograma recursivo siempre necesita dos ingredientes:

caso base al menos una situación, comprobada primero, en la que el subprograma puede responder directamente, sin llamarse a sí mismo de nuevo. Esto es lo que eventualmente detiene la recursión.

caso recursivo la situación en la que el subprograma se llama a sí mismo, pero sobre una versión **más pequeña** del mismo problema —una que tiene garantizado estar más cerca de un caso base que aquella con la que se inició.

Omitir el caso base, o no conseguir que el problema sea más pequeño en cada llamada recursiva, hace que el subprograma se llame a sí mismo indefinidamente —o más bien, hasta que se quede sin **pila**: cada llamada, recursiva o no, necesita su propio almacenamiento privado para sus parámetros y variables locales, apilado sobre las llamadas que están esperando a que termine. Una recursión sin salida eventualmente apila tantas de estas que el programa se bloquea con un **desbordamiento de pila**, la versión de C++ de quedarse sin «ranuras de llamada».

Ya tenemos un ejemplo perfecto a mano: `fact`, de antes en este capítulo. Su propia definición matemática ya es recursiva —$n! = 1$ cuando $n \leq 1$, y $n! = n \times (n-1)!$ en caso contrario— por lo que traducirla a C++ es casi como copiar la fórmula directamente:

```
int fact(int n)                                                    C++
{
    if (n <= 1)
        return 1;              // base case
    return n * fact(n - 1);    // recursive case: a smaller problem
}
```

Compara esto con la versión iterativa que escribimos antes, con su bucle `for` explícito y su acumulador `r`: aquí no se cuenta ni acumula nada a mano. `fact(5)` simplemente dice «5 multiplicado por lo que resulte ser `fact(4)`» y confía en que el resto se resolverá solo, una llamada más pequeña a la vez, hasta llegar a `fact(1)`, que no necesita confiar en nada —simplemente responde.

3·3.2 Ejemplos

Tomemos la suma acumulada que diseñamos en torno a un invariante de bucle antes en este capítulo: `sum == 1 + 2 + ... + N`. Leída como una definición recursiva en lugar de un bucle, la misma idea dice: la suma de los primeros `n` naturales es `0` cuando `n` es `0`, y `n` más la suma de los primeros `n - 1` naturales en caso contrario.

```
int sumUpTo(int n)                                                 C++
{
    if (n == 0)
        return 0;              // base case
```

```
    return n + sumUpTo(n - 1); // recursive case
}
```

Ayuda, las primeras veces, desarrollar una llamada pequeña a mano y observar que nada se calcula hasta que finalmente se alcanza el caso base:

```
sumUpTo(4)                                                    Display
      = 4 + sumUpTo(3)
      = 4 + (3 + sumUpTo(2))
      = 4 + (3 + (2 + sumUpTo(1)))
      = 4 + (3 + (2 + (1 + sumUpTo(0))))
      = 4 + (3 + (2 + (1 + 0)))
      = 10
```

Cada línea espera, sin resolver, a que la llamada anidada dentro de ella responda primero —exactamente la pila de llamadas pendientes descrita anteriormente, con una profundidad máxima de cuatro aquí.

`fact` y `sumUpTo` realizan una única llamada recursiva cada vez, por lo que sus llamadas forman una cadena simple, tan larga como `n`. No toda recursión es tan simple. La secuencia de Fibonacci —cada número es la suma de los dos anteriores— es recursiva por definición, y la traducción directa realiza **dos** llamadas recursivas por llamada:

```
int fib(int n)                                                C++
{
    if (n <= 1)
        return n;                       // base case
    return fib(n - 1) + fib(n - 2);  // two smaller problems
}
```

En lugar de una cadena, las llamadas forman ahora un árbol ramificado: `fib(5)` llama a `fib(4)` y `fib(3)`, `fib(4)` llama a `fib(3)` y `fib(2)`, y así sucesivamente —con `fib(3)` calculado dos veces en el camino, `fib(2)` varias veces más, y la repetición empeorando rápidamente a medida que `n` crece. Esta es la misma lección de Sección 1·4, cuando comparamos por primera vez algoritmos por cómo **crece** su coste: una definición recursiva copiada fielmente en código no es automáticamente un programa eficiente, aunque sea perfectamente correcto.

3·3.3 Tipos de recursión

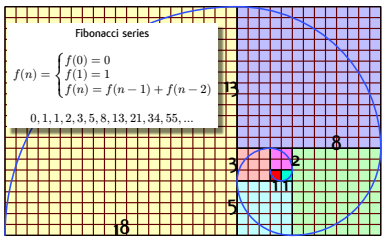
Los subprogramas pueden llamarse a sí mismos de más de una manera. Vale la pena nombrar las diferencias, ya que el vocabulario es suficientemente común como para que lo encuentres de nuevo.

Recursión directa el subprograma se llama a sí mismo, exactamente como hicieron `fact`, `sumUpTo` y `fib` arriba.

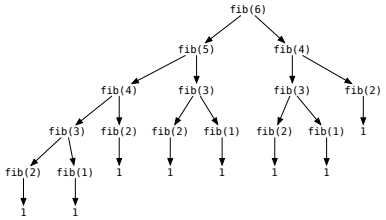
Recursión indirecta un subprograma llama a un segundo, que —quizás después de una cadena de más llamadas— termina llamando al primero de nuevo. El caso más simple son solo dos subprogramas que se llaman entre sí:

```
void a()
{
```

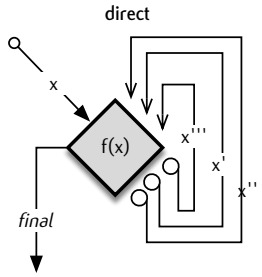
93: La sucesión de Fibonacci es un importante tema de estudio en matemáticas



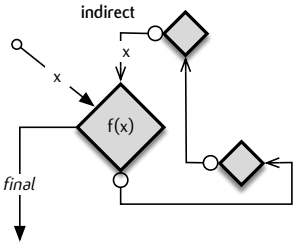
94: Obsérvese que en una solución recursiva a la sucesión de Fibonacci, aunque es la más sencilla de escribir y entender, las **llamadas repetitivas** a la misma función crecen exponencialmente con la profundidad de la recursión



95: Recursión directa: el subprograma se llama a sí mismo



96: Recursión indirecta: una función llama a otra función, que a su vez llama a la primera función



```
  b();
}

void b()
{
  a();
}
```

Como se muestra aquí no hay ningún caso base a la vista, por lo que `a` y `b` se llamarían entre sí indefinidamente. En un programa real, exactamente como con la recursión directa, al menos una de las dos debe eventualmente tomar un camino que omita la llamada a la otra —de lo contrario, el par recurre sin fin, igual que lo haría con toda seguridad un único subprograma al que le faltara su caso base.

Importan dos distinciones más: dónde, dentro del subprograma, se sitúa la llamada recursiva, y cuántas llamadas recursivas realiza.

Recursión de cola la llamada recursiva es la última sentencia que ejecuta el subprograma —no queda nada por hacer una vez que retorna.

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

int main()
{
  void countDown(int n);

  countDown(5);

  return 0;
}

void countDown(int n)
{
  if (n != 0) {
    cout << n << " ";
    countDown(n - 1);   // the last thing this call does
  }
}
```

La recursión de cola es la más fácil de convertir en iteración simple.

Recursión de cabeza la llamada recursiva es la primera sentencia que ejecuta el subprograma —todo lo demás ocurre solo después de que retorna.

```C++
void countUp(int n)
{
  if (n != 0) {
    countUp(n - 1);      // the first thing this call does
    cout << n << " ";
  }
}
```

Los dos parecen casi idénticos, pero `countDown(3)` imprime `3 2 1`, mientras que `countUp(3)` imprime `1 2 3`. En `countDown`, cada llamada imprime **antes** de pasar el control a la siguiente, por lo que los números aparecen en el orden en que se realizan las llamadas. En `countUp`, cada llamada espera a que su llamada recursiva termine completamente antes de imprimir nada, por lo que los números aparecen en orden inverso —la llamada más pequeña, la que se desenrolla primero, imprime primero.

• La recursión de cola tiene otra ventaja: como nada ocurre después de la llamada recursiva, algunos compiladores pueden convertirla en un bucle ordinario entre bastidores, evitando el coste de crecimiento de pila descrito en Sección 3·3.4 más abajo.

Recursión lineal el subprograma realiza exactamente una llamada recursiva cada vez que se ejecuta, como hacen `fact`, `sumUpTo`, `countDown` y `countUp` —sus llamadas forman una cadena simple.

Recursión en árbol el subprograma realiza más de una llamada recursiva cada vez que se ejecuta, como hace `fib` arriba. Sus llamadas forman un árbol ramificado en lugar de una cadena, y el coste puede crecer de forma explosiva: con dos llamadas recursivas en casi cada paso, el número de llamadas realizadas aproximadamente se duplica con cada incremento de `n`, dando la complejidad temporal exponencial que ya señalamos para `fib`.

La recursión puede resolver algunos problemas genuinamente delicados con un código notablemente reducido. Muchas estructuras de datos que encontrarás más adelante en tus estudios —árboles, grafos— se recorren naturalmente con algoritmos recursivos, con una llamada que maneja cada rama.

El rompecabezas de las Torres de Hanói se menciona a menudo como el mejor ejemplo de esa elegancia. Existe una solución iterativa, pero su código no se parece en nada al rompecabezas; la solución recursiva, por el contrario, se lee casi como las propias reglas del rompecabezas.

Imagina un procedimiento que mueve toda una pila de `n` discos de una varilla a otra, usando la tercera varilla como almacenamiento temporal en el camino: `moveGroupUsing(n, base, inter, dest)` mueve `n` discos desde la varilla `base` hasta la varilla `dest`, estacionando discos en la varilla `inter` siempre que necesite un lugar donde dejarlos. La clave recursiva es que mover `n` discos se reduce a tres movimientos más pequeños: mover los `n - 1` discos superiores fuera del camino, desde `base` hacia `inter` (usando `dest` como lugar temporal para ese movimiento más pequeño); mover el único disco restante, el más grande, directamente de `base` a `dest`; y luego mover esos mismos `n - 1` discos desde `inter` hacia `dest` (usando ahora `base` como lugar temporal). Cada «movimiento más pequeño» es exactamente el mismo problema, con un disco menos —lo que lo hace recursivo—, y exactamente lo que el caso base (`n == 0`: nada que mover) finalmente detiene.

```cpp
// hanoi.cpp
// juanfc 2026-09-09
#include <iostream>
using namespace std;

int main()
{
    void moveGroupUsing(int n, int base, int inter, int dest);
    int noDiscs;
    do {
        cout << "No. of discs (0 to end): ";
        cin >> noDiscs;
        moveGroupUsing(noDiscs, 1, 2, 3);
    } while (noDiscs > 0);
    return 0;
}

void moveGroupUsing(int n, int base, int inter, int dest)
{
    if (n > 0) {
        moveGroupUsing(n - 1, base, dest, inter);
        cout << "from " << base << " to " << dest << endl;
```

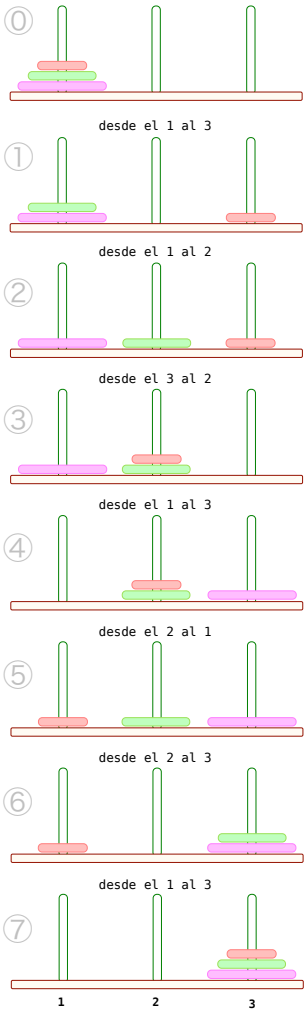
```
        moveGroupUsing(n - 1, inter, base, dest);
    }
}
```

Para dos discos, `moveGroupUsing(2, 1, 2, 3)` imprime:

```
from 1 to 3                                                    Display
from 1 to 2
from 3 to 2
from 1 to 3
from 2 to 1
from 2 to 3
from 1 to 3
```

exactamente la conocida solución de tres movimientos: el disco pequeño fuera del camino, el disco grande al otro lado, el disco pequeño encima.

97: Torres de Hanói



3·3.4 Recursión frente a Iteración

Lo que un bucle puede calcular, un subprograma recursivo también puede calcularlo, y viceversa —los dos son, en ese sentido, igualmente potentes. Lo que difiere es el coste y la claridad, y ambos cortan en direcciones opuestas.

La iteración reutiliza el mismo puñado de variables en cada pasada, por lo que su uso de memoria permanece constante independientemente de cuántas veces se ejecute el bucle; un `for` o `while` que itera un millón de veces no cuesta más almacenamiento que uno que itera diez. La recursión, por el contrario, abre un nuevo marco de pila en cada llamada y solo lo cierra una vez que esa llamada retorna, por lo que su uso de memoria crece con la **profundidad** de la recursión —`fact(20)` abre veinte marcos antes de que el primero pueda cerrarse. Lleva esa profundidad lo suficientemente lejos —un `n` grande, o un caso base que falta— y el programa se bloquea con un desbordamiento de pila mucho antes de que un bucle equivalente hubiera mostrado ninguna tensión. Las llamadas recursivas también conllevan la sobrecarga ordinaria de cualquier llamada a función, pagada una vez por elemento procesado; un bucle no paga tal tarifa.

Frente a ese coste, la recursión ofrece claridad, precisamente cuando la propia definición de un problema es recursiva: `fact` y `sumUpTo` arriba se leen casi exactamente como las definiciones matemáticas que implementan, sin necesitar ninguna variable de contabilidad para llevar un total acumulado a través de las iteraciones. Incluso explica un detalle de antes en el curso: cuando mencionamos por primera vez Quicksort (Figura 28), dijimos que «ordena recursivamente» las sublistas que crea —ahora sabes exactamente lo que eso significaba. Ese algoritmo, y otros similares, dividen un problema en partes más pequeñas del mismo problema —precisamente la forma para la que está construida la recursión— razón por la cual seguirás encontrándola en cuanto las estructuras con las que trabajamos, a partir del próximo capítulo, dejen de ser números individuales.

Para los problemas simples y numéricos que hemos encontrado hasta ahora, sin embargo, la iteración es habitualmente más clara y más barata: sin riesgo de desbordamiento de pila, sin sobrecarga por llamada, y la disciplina del invariante de bucle de antes en este capítulo para razonar

sobre la corrección directamente. Como regla general para este curso: opta por defecto por la iteración, y recurre a la recursión cuando un problema está definido naturalmente en términos de versiones más pequeñas de sí mismo, de modo que la solución recursiva sea dramáticamente más clara que cualquier bucle que pudieras escribir para él.

Esto es exactamente la **recursión de cola** que distinguimos de la recursión de cabeza arriba —el caso en que la llamada recursiva es lo último que hace el subprograma. Es útil saber que existe; en este curso no nos apoyaremos en ella.

3·3.5 Algoritmo general para convertir recursión en iteración

Esta sección es opcional y va más allá del temario. Está aquí para el lector curioso que quiere ver **por qué** la recursión es, en un sentido preciso, más potente que la iteración.

Cualquier función recursiva puede reescribirse como iterativa. La razón es simple: la recursión no es más que el uso automático de una estructura de datos oculta —la **pila de llamadas**. Cada vez que una función se llama a sí misma, la máquina guarda silenciosamente los parámetros actuales, las variables locales y el punto exacto donde reanudar una vez que la llamada retorne. La iteración no hace nada de esto por ti. Por eso el truco consiste en **hacer esa pila explícita** y gestionarla a mano.

La receta universal. Sustituye cada llamada recursiva con la gestión que la máquina hacía en tu nombre:

1. Mantén una `stack` explícita de **marcos**. Cada marco almacena los parámetros guardados, las variables locales guardadas y una **etiqueta de retorno** —un marcador que indica desde qué punto del código continuar una vez que esa llamada termina.
2. Envuelve todo el cuerpo en un único bucle que se ejecuta `while` la pila no está vacía.
3. Convierte cada llamada recursiva en: empuja un nuevo marco (con los valores que aún necesitarás después y la etiqueta de retorno), restablece los parámetros con los argumentos de la llamada, y salta al inicio.
4. Convierte cada `return` en: almacena el resultado, extrae el marco superior, restaura sus valores guardados y salta a su etiqueta de retorno.

Eso es todo. La transformación es completamente mecánica y **absolutamente general** —funciona para toda función recursiva, por enredada que sea— y es, francamente, fea: estás escribiendo a mano la maquinaria exacta que el compilador genera por ti. Esquemáticamente:

```text
push initial frame
while stack is not empty:
    read the top frame, jump to its return label
  L_start:
    ... work ...
    // for a recursive call f(args):
    save (locals, L_after) on the stack
    parameters <- args
    goto L_start
  L_after:
    ... use the returned value ...
    // for "return E":
    result <- E
    pop the frame
    goto (that frame's return label)
```

Obsérvense las **etiquetas de retorno** y los saltos explícitos: este es precisamente el flujo de control de bajo nivel que la elegante versión recursiva oculta —razón exacta por la que nunca escribimos así en código real. (Knuth 1974) trabaja las transformaciones legibles en detalle completo.

El caso especial fácil: la recursión de cola. Cuando la llamada recursiva es lo **último** que hace la función —no queda nada por calcular después de que retorne— no hay trabajo guardado que apilar. El marco es inútil, y todo se reduce a un bucle simple sin pila en absoluto:

```cpp
// tail-recursive
int gcd(int a, int b)
{
    return b == 0 ? a : gcd(b, a % b);
}
// iterative — no stack needed
int gcd(int a, int b)
{
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

Este es el caso que los compiladores optimizan automáticamente (**optimización de llamada de cola**), y el que siempre puedes reescribir a mano sin ninguna estructura auxiliar.

¿Hasta dónde supera realmente la recursión a la iteración? Aquí está el punto profundo. Para la recursión de cola no necesitamos pila; para la receta general sí la necesitamos. ¿Podríamos ser siempre más inteligentes y deshacernos de ella, conservando solo un número fijo de variables ordinarias y un bucle? En general, la respuesta es **no**. Vistos como **esquemas** de programa abstractos, la recursión es estrictamente más potente que la iteración: hay definiciones recursivas que no pueden simularse mediante ningún bucle usando solo una cantidad acotada de memoria extra —la pila auxiliar es genuinamente inevitable ((Paterson, Hewitt 1970)). Así que la pila de nuestra receta no es pereza de nuestra parte; es esencial. Esa memoria ilimitada de tipo último-en-entrar-primero-en-salir es precisamente la potencia adicional que la recursión te compra frente a un simple `for` o `while`.

En la práctica, una vez que aplicas la receta mecánica a una función **específica**, a menudo puedes simplificar el resultado en un bucle idiomático. (Bird 1977) cataloga estas técnicas basadas en pilas sistemáticamente, y (Barron 1968) es el clásico monógrafo breve sobre la recursión y su eliminación. Pero la forma general fea es tu garantía: la conversión es **siempre** posible.

3·3.6 Punto de control

1) Todo subprograma recursivo necesita dos ingredientes. Nómbralos y di de qué es responsable cada uno.
2) ¿Qué ocurre si una recursión no tiene caso base, o si la llamada recursiva no se realiza sobre un problema **más pequeño**? ¿Cuál es el nombre del bloqueo que finalmente causa?
3) `countDown` y `countUp` difieren en una sola línea. ¿Qué imprime cada uno para `n = 4`, y por qué uno de ellos sale en orden inverso?

4) Clasifica cada uno de los siguientes como recursión **directa** o **indirecta**, y como recursión **lineal** o **en árbol**: `fact`, `fib`, y un par de procedimientos `a` y `b` en los que `a` llama a `b` y `b` llama a `a`.

5) El `fib` recursivo es perfectamente correcto, pero muy lento. En términos de la forma que toman sus llamadas, explica por qué su tiempo de ejecución crece tan rápido.

6) Comparado con un bucle equivalente, ¿qué coste extra paga la recursión en memoria, y qué fallo arriesga una recursión muy profunda?

7) ¿Qué tipo de recursión puede convertirse en un bucle simple sin memoria adicional alguna, y por qué? ¿Y por qué, en general, la pila auxiliar **no** puede eliminarse siempre?

8) Escribe un `int power(int base, int exp)` recursivo (supón `exp >= 0`) que devuelva `base` elevado a la potencia `exp`. Señala su caso base y su caso recursivo.

Referencias

BARRON, David W., 1968. *Recursive Techniques in Programming*. New York: American Elsevier. Computer Monographs.

La monografía breve clásica sobre la recursión y su eliminación.

BIRD, Richard S., 1977. Notes on Recursion Elimination. *Communications of the ACM*. 1977. Vol. 20, no. 6, p. 434-439. DOI **10.1145/359605.359630**.

Un catálogo sistemático de métodos basados en pilas para eliminar la recursión, comparados por coste de ejecución.

KNUTH, Donald E., 1974. Structured Programming with go to Statements. *ACM Computing Surveys*. 1974. Vol. 6, no. 4, p. 261-301. DOI **10.1145/356635.356640**.

El relato definitivo sobre cómo convertir la recursión en iteración a mano, trabajando paso a paso las transformaciones legibles. A pesar del título provocador, el artículo defiende una buena estructura del programa, no el uso de `goto`.

PATERSON, Michael S. y HEWITT, Carl E., 1970. Comparative Schematology. In: *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*. New York: ACM. 1970. p. 119-127.

Demuestra, al nivel de esquemas abstractos de programas, que la recursión es estrictamente más potente que la iteración: en general, la pila auxiliar no puede eliminarse.

Tarea 6: Subprogramas

Asegúrate de haber completado todos los ejercicios de la Tarea 5 antes de comenzar esta. A partir de este punto, la mayoría de los problemas deben resolverse usando subprogramas. La estructura típica es: `main()` llama a un subprograma, que a su vez llama a otros. La mayoría de los subprogramas deben realizar cálculos o modificar parámetros pasados por referencia — **no deben leer con `cin` ni imprimir con `cout`** salvo que estén específicamente diseñados para ello.

Los problemas marcados con 🧊 son desafiantes; no dudes en pedir ayuda. Los problemas marcados con 🧊🧊 están pensados solo para los estudiantes más determinados.

1 Supón que las siguientes variables están declaradas al comienzo de `main()`:

```C++
int a, b, c;
bool theEnd;
```

y que los siguientes prototipos de subprogramas están disponibles:

```C++
bool one  (int  x, int y);
void two  (int& x, int y);
int  three(int  x);
```

¿Cuáles de las siguientes llamadas desde dentro de `main()` son legales? Para las que no lo sean, explica por qué.

1. `if ( one(a, b) ) {`,
2. `two(a, b+3)`,
3. `theEnd = one(c, 5)`,
4. `theEnd = two(c, 5)`,
5. `two(a, three(a))`,
6. `two(three(b), c)`,
7. `if (three(a)) {`,
8. `b = three(two(a, 5))`,
9. `two(4, c)`,

Escribe un archivo `t06e01.answers.txt` con cada una de las llamadas anteriores seguidas de tus comentarios.

2 Construye una función que devuelva el número de dígitos necesarios para representar `n` en una `base` dada: › Sol

```
int numDigits(int n, int base=10);
```

Desde `main()`, llámala con `n = 2021` en base `10` y base 2. Debe devolver `4` y `11` respectivamente ($2021 = 11111100101_2$). `numDigits(0)` debe devolver `1`.

3 Construye una función que devuelva el p-ésimo dígito de `n` en la base `base`: › Sol

```
int digit(int n, int p, int base);
```

Los dígitos se cuentan desde la derecha, empezando en `0`. Por ejemplo, `digit(25064, 1, 10)` devuelve `6` y `digit(25064, 10, 10)` devuelve `0`.

4 Construye una función que calcule la n-ésima potencia entera de un número real:

```
float powx(float x, int n);
```

Calcúlala por multiplicación repetida (`r *= x`, comenzando desde `1`). El exponente `n` puede ser negativo, cero o positivo. Desde `main()`, pide al usuario `x` y `n` e imprime el resultado. › Sol

5 Construye un procedimiento que resuelva la ecuación cuadrática $ax^2 + bx + c = 0$ y devuelva sus dos raíces mediante parámetros por referencia:

```
void quadEq(float a, float b, float c, float& x1, float& x2);
```

Las raíces son $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. Desde `main()`, pide a, b, c e imprime `x1` y `x2`. Supón que el discriminante $d = b^2 - 4ac$ es positivo y $a \neq 0$.

Para una solución más completa que maneje todos los casos, añade un parámetro de salida entero `type` que codifique el tipo de solución encontrada: › Sol

1. dos raíces reales distintas,
2. partes real e imaginaria de un par complejo conjugado,
3. una raíz real doble,
4. una sola raíz (porque $a = 0$),
5. sin solución ($a = b = 0$, $c \neq 0$),
6. cualquier x es solución ($a = b = c = 0$),

Construye:

```
void quadEq(float a, float b, float c, float& x1, float& x2, int& type);
```

y pruébala con:

```
// (type 1) 2 roots: 1 and -2
quadEq( 1,  1, -2, x1, x2, type);
// (type 3) 1 double root
quadEq( 1, -2,  1, x1, x2, type);
// (type 4) 1 root: 0.5 only
quadEq( 0, -2,  1, x1, x2, type);
// (type 5) impossible
quadEq( 0,  0,  1, x1, x2, type);
// (type 2) complex: +- i
quadEq( 1,  0,  1, x1, x2, type);
// (type 6) any x
quadEq( 0,  0,  0, x1, x2, type);
```

6 Un año es **bisiesto** si es divisible por 4 pero no por 100, o si es divisible por 400. Por ejemplo, 2016 y 2000 son años bisiestos; 1800, 1900, 2100 y 2300 no lo son. Construye una función que reciba un año y devuelva si es bisiesto. Desde `main()`, pide al usuario un año e imprime `"año bisiesto"` o `"no es año bisiesto"`. › Sol

7 Construye una función `double harmonic(int n)` que devuelva la suma de los primeros `n` términos de la serie armónica: › Sol

$$\sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots.$$

Desde `main()`, pide `n` e imprime el resultado. Verifica que la suma crece sin límite a medida que `n` aumenta.

8 Construye una función `double alternate(int n)` que devuelva la suma de la serie armónica alternada: › Sol

$$\sum_{i=1}^{n} (-1)^{i+1} \frac{1}{i} = \frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots.$$

Desde `main()`, llámala con un `n` dado e imprime el resultado. Verifica que converge a $\ln 2 = 0.6931471805\ldots$

9 Construye una función `float gravity(float m1, float m2, float d)` que devuelva la fuerza gravitatoria entre dos masas m_1 y m_2 separadas por una distancia d: › Sol

$$F_g = -G \frac{m_1 m_2}{d^2}$$

donde $G = 6.67428 \times 10^{-11}$ N $\frac{\text{m}^2}{\text{kg}^2}$ (constante de Cavendish).

Úsala para calcular:

1. La aceleración gravitatoria en la superficie de la Tierra ($m_1 = 5.974 \times 10^{24}$ kg, $d = 6,378,140$ m, $m_2 = 1$ kg),
2. La aceleración en órbita geoestacionaria (**35.786 km** sobre la superficie),
3. La gravedad superficial del asteroide Bennu (radio 245 m, masa 7.329×10^{10} kg)

10 Usando la función de año bisiesto del ejercicio 6, construye una función que reciba una fecha completa (año, mes, día) y devuelva si es una fecha de calendario válida. Desde `main()`, pide al usuario año, mes y día, e imprime `"fecha válida"` o `"fecha no válida"`. › Sol

11 Construye una función `bool isPrime(int n)` que devuelva `true` si `n` es primo. Empieza probando todos los divisores (`i`) desde 2 hasta `n`; después optimiza deteniendo `i` en $\sqrt{n}$ y omitiendo los divisores pares. › Sol

HINT: En lugar de calcular $\sqrt{n}$, lo que requiere un cálculo en serie de punto flotante de la biblioteca `<math>`, simplemente comprueba `d*d < n and n%d != 0` en cada iteración.

12 Usando `bool isPrime(long long n)` de los ejercicios anteriores, implementa una función `unsigned long long nPrime(int n)` que devuelva el n-ésimo número primo. Pruébala con: › Sol

```cpp
int main()
{
    unsigned long long nPrime(int);

    for (int i = 1; i < 100; i += 10)
        cout << i << ": " << nPrime(i) << endl;
    return 0;
}
```

Salida esperada:

```
 1:   2                                                          Display
11:  31
21:  73
31: 127
41: 179
51: 233
61: 283
71: 353
81: 419
91: 467
```

Después pruébala calculando la suma de los primeros `1000` números primos. Finalmente, determina e indica si esa suma es prima.

o O o

Tarea 7: Subprogramas++

Asegúrate de haber completado todos los ejercicios de la Tarea 6 antes de comenzar esta. A partir de aquí, la mayoría de los problemas deben resolverse usando subprogramas. La estructura típica es: `main()` llama a un subprograma, que a su vez llama a otros. La mayoría de los subprogramas deben realizar cálculos o modificar parámetros pasados por referencia — **no deben leer con `cin` ni imprimir con `cout`** salvo que estén específicamente diseñados para ello.

Los problemas marcados con 🧊 son desafiantes; no dudes en pedir ayuda. Los marcados con 🧊🧊 están pensados solo para los estudiantes más determinados.

1 Escribe un procedimiento sencillo sin parámetros que devuelva la longitud de la secuencia más larga introducida por teclado en la que los caracteres `0` y `1` permanecen en orden ascendente. Por ejemplo:

```
10100001102 -> 6 corresponding to 000011                    example
00112 -> 4 no disorder
112 -> 2 no disorder
002 -> 2 no disorder
2 -> 0 nothing
```

El procedimiento (por ejemplo: `int lss(void);`) devolverá esta longitud, que representa la secuencia ascendente más larga, hasta que se lea un `2`, y `main()` imprimirá ese número. › Sol

2 🧊 Usando `digitAtp()` del ejercicio 1, construye un procedimiento `void printSumSymDigits(int n)` que imprima las sumas de los pares de dígitos colocados simétricamente. Por ejemplo, `2354869` produce: › Sol

```
2354869: 11 9 13 4                                          example
6582: 8 13
121: 2 2
0: 0
```

donde `11 = 2+9`, `9 = 3+6`, `13 = 5+8`, `4` (dígito central, sin pareja). También necesitarás una función auxiliar `int noDigits(int n)`.

HINT: Los subprogramas no deben leer ni imprimir (sin `cin`/`cout`) salvo que estén específicamente diseñados para ello. Aquí conviene reutilizar `digitAtp()` y `noDigits()`. Un posible esqueleto:

```cpp
int main()                                                  C++
{
    void printSumSymDigits(int);

    printSumSymDigits(2354869);

    return 0;
}

void printSumSymDigits(int n)
{
    int digitAtp(int, int);
    int noDigits(int);

    // cout << ...
}

int digitAtp(int n, int pos) { /* ... */ return ...; }
int noDigits(int n)          { /* ... */ return ...; }
```

3 🧊 Construye una función `bool isPalindromic(int n)` que devuelva `true` si `n` se lee igual de izquierda a derecha que de derecha a izquierda (p. ej., `898`, `1719171` y `1` son palíndromos; `13` y `123` no lo son). Reutiliza `digitAtp()` y `noDigits()` de los ejercicios anteriores. › Sol

HINT: Compara el dígito `i` con el dígito `l-i`, iterando `i` desde `0` hasta `l/2` mientras los dígitos simétricos coincidan, donde `l` es el número de dígitos. Usa un bucle `while`: debes detenerte en cuanto encuentres una discrepancia, y un bucle `for` requeriría un `break` en el cuerpo o una variable guardia para conseguirlo. Incorrecto si se usa en este curso.

4 🧊 Construye una función `bool isPalindromic(int n)` que devuelva `true` si `n` se lee igual de izquierda a derecha que de derecha a izquierda (p. ej., `898`, `1719171` y `1` son palíndromos; `13` y `123` no lo son). Construye `digitAtp()` usando la función `powerint()` › Sol

5 🧊 Construye un **procedimiento** `void primeDecomposition(int n)` que imprima la descomposición en factores primos de `n`. Por ejemplo: $1200 = 2^4 \times 3^1 \times 5^2$. › Sol

HINT: No es necesario llamar a `nPrime()`. En su lugar, intenta dividir `n` por cada entero a partir de 2, contando cuántas veces divide a `n` con un doble bucle y reduciendo `n` cada vez hasta que caiga por debajo de 2. Imprimimos los resultados a medida que los encontramos porque todavía no disponemos de ninguna estructura de datos para almacenarlos.

6 🧊 La conjetura de Goldbach afirma que todo número par mayor que 2 es la suma de dos números primos. Construye un procedimiento `void checkGoldbach(int from, int to)` que verifique esto para todos los números pares del intervalo `[from, to]`. Para los números pares justo superiores a 2: › Sol

```
 4 = 2 + 2                                                    example
 6 = 3 + 3
 8 = 3 + 5
10 = 3 + 7
12 = 5 + 7
```

Si todos los números del rango satisfacen la conjetura, imprime: `"From 20 to 2000 they follow Goldbach's"`. Si alguno falla (por ejemplo `3232`), detente e imprime `"3232 DOES NOT FOLLOW Goldbach's"`.

7 El máximo común divisor (MCD) de dos números naturales es el mayor número natural que divide a ambos. Por ejemplo, $\gcd(8, 12) = 4$. Construye `int gcd(int a, int b)` en tres pasos:

1. **Fuerza bruta.** Construye `int gcdBruteForce(int a, int b)`: comienza desde el menor de `a` y `b` y prueba cada entero hacia abajo hasta encontrar uno que divida a ambos. › Sol
2. **El algoritmo de Euclides** utiliza las propiedades:

$$\begin{cases} \gcd(a, b) = \gcd(b, a - b) \text{ if } a > b \\ \gcd(a, b) = \gcd(b, a) \\ \gcd(a, 0) = a \end{cases}$$

› Sol Por ejemplo: $(12, 18) \to (18, 12) \to (6, 12) \to (12, 6) \to (6, 6)$, por lo que $\gcd(18, 12) = 6$. Una implementación basada en restas:

```cpp
int gcd(int a, int b)
{
    do {
        if (a < b)
            swap(a, b);
        a -= b;
    } while (b != 0);
    return a;
}
```

Ahora reescríbela sustituyendo las restas por el operador `%` para obtener una versión más rápida.

Construye un procedimiento `void simplify(int num, int den, int& nnum, int& nden)` que reduzca la fracción `num/den` a su mínima expresión usando `gcd()`, devolviendo el resultado en `nnum` y `nden`. Pruébalo desde `main()`: › Sol

```cpp
int main()
{
    void testIt(int, int);

    testIt(12,   18);  // 12/18:   2/3
    testIt(54,   24);  // 54/24:   9/4
    testIt(12,   30);  // 12/30:   2/5
    testIt(1260, 600); // 1260/600: 21/10
    return 0;
}
```

8 Construye `int lcmBruteForce(int a, int b)` para el mínimo común múltiplo por fuerza bruta: partiendo del mayor de `a` y `b`, busca hacia arriba el primer número que sea múltiplo de ambos. › Sol

9 Construye `int lcm(int a, int b)` usando la identidad (ver aquí): › Sol

$$\operatorname{lcm}(a, b) = a \cdot b / \gcd(a, b)$$

10 Un método para aproximar $\sqrt{x}$ usa la identidad:

$$\sqrt{n} = \sqrt{p + q} \approx \sqrt{p} + \frac{q}{2\sqrt{p}}$$

La idea es encontrar el mayor cuadrado perfecto $p \leq n$ y aplicar la fórmula. Por ejemplo, si $n = 968$: como $\sqrt{961} = 31$, obtenemos $\sqrt{968} \approx 31 + \frac{7}{2 \cdot 31} = 31.112$. (El valor exacto es $31.11270\ldots$)

Construye `int prevSq(int n)` que devuelva el entero p tal que $p^2 \leq n < (p+1)^2$. Por ejemplo, `prevSq(968)` devuelve `31`. › Sol

11 En un fichero no ejecutable `14.rec2iter.cpp`, diseña el equivalente iterativo del siguiente procedimiento recursivo: › Sol

```cpp
bool f(int n);
int  g(int n);

void rec(int n)
```

```
{
    if (not f(n)) {
        // any statements that do not modify n
        rec(g(n));
    }
}
```

12 Diseña un procedimiento recursivo que lea desde el teclado una secuencia de caracteres de longitud arbitraria, terminada por un punto (`'.'`), e imprima la secuencia en orden inverso. El carácter punto no se imprime; sirve únicamente como terminador.

El procedimiento no debe tener parámetros. `main()` probará el procedimiento. Por ejemplo, si introduces `hola` desde el teclado, la salida debe ser `aloh`. › Sol

13 El **máximo común divisor** (MCD) de dos números naturales p y q es el mayor entero que divide a ambos. El **algoritmo de Euclides** resta repetidamente el valor menor del mayor hasta que ambos son iguales. Por ejemplo, partiendo de $p = 18$, $q = 12$: › Sol

```
18, 12  (p > q  =>  p -= q  =>  p = 6)
 6, 12  (q > p  =>  q -= p  =>  q = 6)
 6,  6  (p == q  =>  gcd = 6)
```
Display

Implementa una versión **recursiva**:

```
int gcd(int p, int q);
```

14 Copia, compila y ejecuta el siguiente programa, que pide repetidamente al usuario dos enteros, realiza la división entera y gestiona la división por cero con un bloque `try`/`catch`: X

```
// t07e14.trycatch.cpp
#include <iostream>
#include <stdexcept>
using namespace std;

int main()
{
    int dividend, divisor;
    cout << "Enter dividend and divisor (0 0 to quit): ";
    cin >> dividend >> divisor;
    while (dividend != 0 or divisor != 0) {
        try {
            if (divisor == 0)
                throw invalid_argument("Divisor cannot be zero");
            cout << dividend << " / " << divisor
                 << " = " << dividend / divisor
                 << " remainder " << dividend % divisor << endl;
        } catch (invalid_argument& e) {
            cout << "Error: " << e.what()
                 << ". Please do not enter zero as the divisor." << endl;
        }
        cout << "Enter dividend and divisor (0 0 to quit): ";
        cin >> dividend >> divisor;
    }
    return 0;
}
```
C++

Tarea 8: Pre-estructuras

Asegúrate de haber completado todos los ejercicios de la Tarea 7 antes de comenzar esta. Como todavía no hemos introducido las estructuras de datos, ninguna de las soluciones siguientes debe usar arrays ni ningún otro tipo estructurado. Podrás usarlos libremente a partir de la Tarea 9.

1 Construye el programa `t08e01.readnum.cpp`. Queremos leer un número del usuario, pero aceptarlo solo si se encuentra dentro de un rango dado. Si el número está fuera de rango, seguir preguntando hasta que se introduzca uno válido. Construye la función: › Sol

```
int readNum(int from, int to);
```

que devuelva un entero introducido por el usuario, rechazando cualquier valor fuera de `[from, to]`. Llámala desde `main()` e imprime el resultado.

2 Construye el programa `t08e02.difextremes.cpp` *(no se necesitan subprogramas)*. Lee enteros del usuario hasta que se introduzca `0`; el `0` no forma parte de la secuencia. Imprime la diferencia entre el valor máximo y el mínimo. Supón que se introduce al menos un número antes del `0`. › Sol

```
1 2 3 4 0    -> 3                                    example
-2 2 0       -> 4
2 2 0        -> 0
-2 0         -> 0
```

3 Construye una función `bool lowUpBalanced()` que lea caracteres del usuario (los espacios se omiten automáticamente con `cin >>`). La entrada termina cuando se introduce un punto `'.'`. La función devuelve `true` si el número de letras mayúsculas es igual al número de letras minúsculas, y `false` en caso contrario. Usa el siguiente `main()`: › Sol

```
int main()                                                C++
{
    bool lowUpBalanced();

    if (lowUpBalanced())
        cout << "Equal number of lower and upper chars" << endl;
    else
        cout << "No equal number of lower and upper chars" << endl;
    return 0;
}
```

4 Dos números son amigos cuando la suma de los divisores propios de cada uno es igual al otro. Construye la función `areAmicable()` que recibe dos números y devuelve si son amigos. Desde `main()` llama a un procedimiento `testAmicables()` que recibe dos números e imprime el resultado. Imprime también todos los pares de números amigos menores que `3000`. › Sol

```
int main()                                                C++
{
    void amicLessThan(int n);
    void printRes(int a, int b);

    printRes(284, 220);
    printRes(121, 212);

    amicLessThan(3000);
    return 0;
```

```
}

void amicLessThan(int n)
{
    bool amicable(int a, int b);
    int addDivisors(int n);
    for (int i = 1; i <= n; ++i)
        for (int j = 0; j < i; ++j)
            if (amicable(i, j))
                cout << i << " " << j << endl;
}

void printRes(int a, int b)
{
    bool amicable(int a, int b);

    cout << "The numbers " << a << " and " << b <<
     (amicable(a, b)? " ARE ": " are NOT ") << "amicable" << endl;
}
```

5 Construye el programa `t08e05.difgreatests.cpp`. Lee enteros del usuario hasta que se introduzca `0`; el `0` no forma parte de la secuencia. Imprime la diferencia entre el mayor y el segundo mayor valor. Supón que se introducen al menos dos números antes del `0`. › Sol

```
-3 4 -22 0          -> 7                                    example
1 2 0               -> 1
2 2 0               -> 0
-3 5 -3 8 6 9 0     -> 1
```

6 Construye el programa `t08e06.summing.cpp` *(no se necesitan subprogramas)*. Lee enteros del usuario de uno en uno, acumulando su suma. Para en cuanto la suma acumulada supere `max = 10`. No incluyas el número que hace que la suma supere el límite. Imprime la suma final. › Sol

o O o

Part 4 Tipos de Datos Estructurados

Contenido de esta parte Contenido del libro

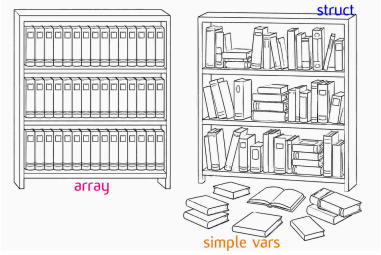
4·1 Paso de Parámetros con Tipos Estructurados 139
4·2 Arrays 140
▸ Tarea 9: Arrays 149
▸ Tarea 10: Arrays++ 156
4·3 Cadenas de caracteres 161
▸ Tarea 11: Cadenas de caracteres 166
4·4 Estructuras o registros 175
▸ Tarea 12: Estructuras 180
4·5 Matrices 185
▸ Tarea 13: Matrices 189
▸ Tarea 14: Problemas de exámenes finales 195

Hasta ahora solo hemos visto tipos de variable capaces de almacenar un único dato — es decir, cuyo contenido no puede descomponerse en partes con significado propio. A estos tipos se les llama también *monolíticos*.

Las variables simples son suficientes para resolver muchos algoritmos, pero los ordenadores existen para algo más que analizar el comportamiento de un puñado de valores. Las tareas que más importan suelen involucrar colecciones de datos: estadísticas sobre listas de registros complejos, o el procesamiento de datos naturales — sonido, lecturas meteorológicas, imágenes — que a menudo implica manejar enormes cantidades de puntos de datos individuales.

Incluso programas modestos que trabajan con datos tratados de manera similar, unas pocas docenas de valores por ejemplo, se vuelven engorrosos con variables simples: habría que repetir la misma acción, línea tras línea, una vez por cada variable. Por eso son tan importantes las estructuras de datos — composiciones de variables simples, colecciones de datos referenciadas por un único nombre colectivo —. Y la más importante de todas ellas es el array.

98: Las variables simples son independientes y deben accederse individualmente, mientras que los tipos de datos estructurados pueden manipularse como un todo mediante un único identificador



4·0.1 Plantillas

Los tipos de estructuras que usaremos en este curso — arrays, cadenas de caracteres, registros y arrays de arrays, o matrices — serán proporcionados por la biblioteca estándar de plantillas de C++ (STL), en lugar del enfoque clásico al estilo C que podrían haber usado cursos más antiguos. Apoyarse en la STL significa apoyarse en un compilador de C++ genuino en lugar de un compilador de C simple, y significa usar un aspecto del lenguaje aún más avanzado: las instancias de **plantillas**

Las plantillas en C++ son un mecanismo que permite escribir código genérico — código que funciona con muchos tipos diferentes sin necesidad de reescribir la misma función para cada uno.

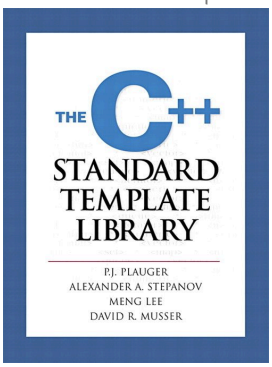
La sobrecarga de funciones permite reutilizar un nombre, pero aún hay que escribir cada cuerpo a mano, aunque sean idénticos salvo por el tipo:

```
void mySwap(int& a, int& b)       { int t = a; a = b; b = t; }
void mySwap(float& a, float& b)   { float t = a; a = b; b = t; }
void mySwap(char& a, char& b)     { char t = a; a = b; b = t; }
```

```
void mySwap(string& a, string& b) { string t = a; a = b; b = t; }
```

y así sucesivamente para cada tipo que pueda necesitarse para intercambiar. Con plantillas, una única definición es suficiente:

99: STL: C++ Standard Template Library




```
template <typename T>
void mySwap(T& a, T& b) { T t = a; a = b; b = t; }
```

El compilador genera la versión para cada tipo que realmente se use.

Tras la declaración de la plantilla anterior, se puede usar directamente

```
int a = 1, b = 2;
mySwap(a, b);          // T = int

string s = "x", t = "y";
mySwap(s, t);          // T = string
```

el compilador deduce el tipo y construye el código necesario para cada llamada que haga nuestro programa, ni una más ni una menos.

Así pues, nos vamos a apoyar en la STL, lo que significa apoyarse también en plantillas y clases. Pero no hay que preocuparse: solo necesitaremos **usar** dos de ellas, la clase `string` y la clase `array` (construida ella misma con plantillas) — y usarlas, en vez de escribir las propias, mantiene la sintaxis agradablemente sencilla, como veremos en cuanto estudiemos `array` y `string` en las secciones que siguen.

4·1 Paso de Parámetros con Tipos Estructurados

Un aspecto importante que cubrimos poco después de introducir los tipos simples, en Sección 2·2, fue el paso de parámetros — valores y referencias — a subprogramas (Sección 3·2). Esa es exactamente la primera pregunta que queremos abordar aquí: ¿cambia algo cuando los parámetros son estructurados en lugar de simples?

La principal ventaja de usar plantillas para arrays y cadenas (las cadenas también son plantillas; para caracteres normales no es necesario usar el constructor de plantilla, pues ya está proporcionado) es que se vuelven **copiables**. Esto nos importa considerablemente, ya que ya sabemos que pasar parámetros por copia es nuestro método predeterminado y seguro. Además, el paso de parámetros no es el único contexto que se beneficia: devolver una estructura basada en la STL desde un subprograma funciona de la misma manera — nunca hay que pensar en que se trata de una estructura. Se copia exactamente igual que si fuera un tipo de dato simple.

En el C clásico, los arrays se declaran de una manera completamente diferente — ~~`int a[3];`~~ — y no son clases: no son copiables, por lo que pasarlos o devolverlos requeriría un mecanismo completamente diferente. Una vez pasados a un subprograma, **degeneran** en un puntero simple, perdiendo en el proceso toda memoria de su propio tamaño. `array`, al ser una clase apropiada, no sufre ninguno de estos problemas.

Todos los tipos basados en la STL son clases, y las clases — como el `struct` de C del que descienden — son copiables por defecto. Esta copiabilidad deriva en última instancia de esta propiedad, y es la razón por la que todos ellos, `string` y `array` incluidos, son copiables.

En resumen, pasar parámetros estructurados no cambiará ninguno de nuestros hábitos: mientras nos ciñamos a estructuras basadas en la STL,

todo lo que aprendimos en Sección 3·2 sigue aplicándose exactamente igual que con los tipos simples.

Algunos programadores, deseosos de optimizar siempre que una estructura pueda ser grande — algo que no podemos saber y no necesitamos preocuparnos en este curso — pasan los tipos de datos estructurados como `const type& structuredParam`. Esto no es incorrecto, y mucha literatura de C++ lo fomenta activamente; sin embargo, complica las interfaces y ayuda poco a aprender a programar en general. Te lleva a los detalles más intrincados del lenguaje C++, algunos de los cuales cambian de versión en versión y no se trasladan a otros lenguajes en absoluto. Por tanto, en nuestros ejemplos, no usaremos este estilo `const&` de paso de parámetros.

4·2 Arrays

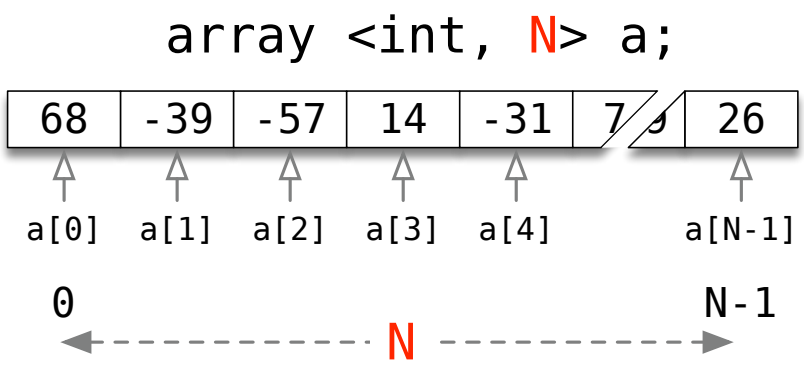
> Los arrays son las estructuras de datos más importantes en la informática por su capacidad para manejar grandes volúmenes de datos de manera eficiente y directa

Los arrays son la base de las cadenas de caracteres y la estructura de datos más importante en los lenguajes de programación. De hecho, podemos pensar en la RAM de un ordenador como un enorme array de bytes — y, desde el punto de vista del hardware, esto es literalmente cierto. A partir de ahora definiremos los arrays y aprenderemos a declararlos y manejarlos; después extenderemos lo aprendido aquí a las cadenas de caracteres (Sección 2·9.1), ya que las cadenas no son más que arrays especializados de caracteres. Finalmente, una vez introducidas las estructuras, volveremos a los arrays para construir matrices.

4·2.1 Definición. El Tipo de Dato Array

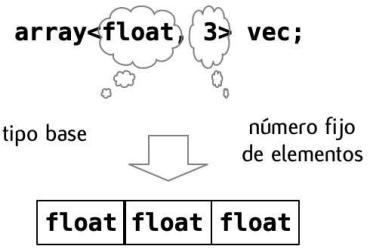
En general

> Un array puede almacenar y trabajar con múltiples valores del mismo tipo de dato,
> indexados por un índice entero que comienza en `0`
>
> Un array en C++ es la repetición del mismo tipo base `N` veces, donde `N` es una constante conocida de antemano:
> `array<baseType, N> arrayName;`



El hecho de que `N` sea una constante implica directamente que el array es una estructura **estática**: no puede crecer ni encoger. El número de elementos es algo que el programador debe decidir en el momento en que se escribe el programa. Esta es la restricción más severa de los arrays — una que otros tipos de la STL como `vector` no comparten — pero es precisamente este tamaño fijo y conocido lo que hace que la sintaxis de un array sea tan conveniente a la hora de construir arrays multidimensionales (matrices), algo muy importante en programación.

100: La sencilla declaración de una sola línea de un array implica crear muchas variables al mismo tiempo



No olvides incluir `#include <array>`, la cabecera donde se define la plantilla `array`. Si se olvida, los errores que emite el compilador son desconcertantes y difíciles de relacionar con el problema real — hay que tener cuidado con esto.

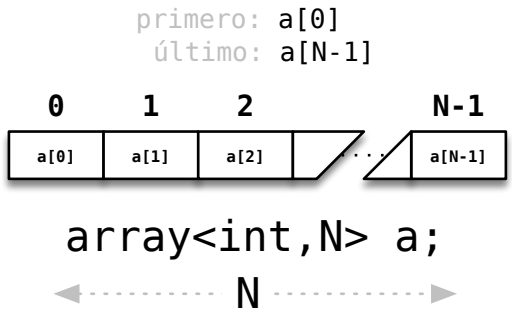
En C++, los arrays se disponen en memoria como una secuencia de variables `baseType`, una justo después de la otra, y cada una de ellas se referencia simplemente por su posición — una posición que comienza en `0`. Por tanto, en sentido estricto, no es realmente un «índice» lo que localiza cada variable, sino el desplazamiento (shift) desde el origen del array en memoria.

El consumo de memoria de un array depende de su tipo de dato y del número `N` de elementos que contiene.

4·2.2 Acceso a los elementos del array

Aunque un array tiene un único nombre, sus `N` elementos pueden accederse y manipularse como variables individuales. Esto es posible porque a cada elemento se le asigna un índice numérico, conocido como subíndice. El subíndice identifica un elemento específico dentro del array. El primer elemento tiene subíndice `0`, el segundo elemento tiene subíndice `1`, y así sucesivamente, siendo el último elemento el que tiene subíndice `N-1`.

Así pues, para acceder a un elemento de un array necesitamos tanto el nombre del array como un entero simple: un índice (desplazamiento) que empieza en `0` y termina en el número de elementos menos 1 (`N-1`).



Cada posición es una variable **independiente**:

```C++
#include <array>
int main()
{
  array<int, 10> a;
  a[0] = 33;
  cout << a[9] << endl; // will print garbage
  return 0;
}
```

Para acceder a **todos** los elementos — para recorrer el array — `for` es el bucle de elección, ya que ya sabemos de antemano exactamente cuántas iteraciones necesitamos (`N`). Pero `for` solo es la herramienta adecuada cuando pretendemos recorrer todo el array sin detenernos a mitad de camino; si en cambio podríamos necesitar detenernos antes — por ejemplo, en cuanto encontremos lo que buscamos — `while` sigue siendo la elección correcta, siguiendo la misma regla `for`/`while` que ya establecimos.

```cpp
#include <array>
int main()
{
  array<int, 10> a;
  for (int i = 0; i < 10; ++i) {
      a[i] = i*i;
  }
  for (int i = 0; i < 10; ++i) {
      cout << a[i] << ", ";
  }
  cout << endl;
  return 0;
}
```

Una advertencia sobre `[]`. El operador de subíndice `[]` **no** comprueba si el índice está dentro del rango. Como la RAM realmente es un enorme array, un acceso fuera de rango como `a[10]` sobre un `array<int, 10>` simplemente lee o escribe los bytes que se encuentran junto a tu array. Este es un **comportamiento indefinido**: el programa puede bloquearse, imprimir disparates, o — lo peor de todo — parecer que funciona hoy y fallar mañana. Mantener cada índice dentro de `0 .. N-1` es responsabilidad exclusiva del programador.

Por este motivo, `std::array` también ofrece un acceso **con comprobación de límites**, `.at(i)`, que verifica el índice y lanza una excepción `std::out_of_range` en lugar de corromper silenciosamente la memoria. Es ligeramente más lento que `[]`, pero mucho más seguro mientras se aprende o se depura:

```
array<int, 10> a;
a[10]    = 5;   // undefined behaviour: no check at all
a.at(10) = 5;   // throws std::out_of_range
```

4·2.3 Operaciones con Arrays

Los arrays tienen sus propias operaciones asociadas. Los parámetros de tipo array requieren que los parámetros formales y actuales tengan exactamente el mismo tipo: mismo tipo de elemento y mismo tamaño. No es necesario declarar un tipo con nombre para esto, porque cada aparición de `array<float, N>` denota el mismo tipo. Sin embargo, definir un alias hace el código más fácil de leer y mantener, y permite cambiar el tamaño en un solo lugar. Por esta razón, siempre declararemos los arrays como nuevos tipos con nombre usando `typedef` antes de usarlos:

```
const int N = 3;
const int NPER = 55;
typedef array<float, N> TVector;
typedef array<int, NPER> TAges;
```

Pero hay que tener cuidado: si no se da ningún valor inicial a un array local, su contenido es indeterminado (basura), como ocurre con las variables simples.

Para enviar un array completo como literal, podemos construir un temporal del tipo exacto del array:

```
printVec(TVec{0,1,2,3,4});   // N must be 5 here
```

El número de valores no puede superar `N`; se permiten menos valores, y el resto se pone a `0`.

Los tamaños de los arrays son constantes en tiempo de compilación. La mejor práctica es declararlos como `const`antes, como en los ejemplos anteriores. También podemos preguntar al array sobre su tamaño con la función miembro `.size()`. A diferencia de un array de C simple, `std::array` es una clase, pero no almacena datos extra: el tamaño es parte de su tipo.

```
const int NPER = 55;                                                C++
typedef  array<int, NPER> TAges;
int main()
{
  TAges ages;
  cout << ages.size() << endl; // -> 55
}
```

Usar `N` o `v.size()` en la condición del bucle es igualmente correcto — pero nótese que `.size()` documenta el código **menos** que un nombre de constante bien elegido como `N` o `MAX_NUMBER`, ya que el nombre de la constante le dice al lector qué significa el número.

Comparación de arrays: No solo se puede asignar un array a otro (del mismo tipo), también se pueden comparar:

```
int main()                                                          C++
{
  TVec a, b;
  a = b;
  bool eq = (a == b);
  bool ne = (a != b);
  bool gt = (a > b);
  ...
}
```

Para entender las comparaciones `>`, `<`, `<=` y `>=`, piensa en la forma en que las palabras se ordenan en un diccionario (orden lexicográfico): los dos arrays se comparan celda a celda, comenzando en el índice `0`; en cuanto se encuentra una celda que difiere (mayor o menor) de la celda correspondiente en el otro array, la comparación se detiene ahí y esa diferencia decide el resultado.

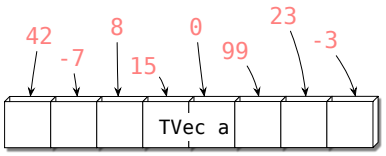
4·2.3.1 Inicialización rápida

Asignar valores iniciales a variables simples en la declaración ha demostrado ser muy práctico. No solo evita la omisión accidental de valores iniciales críticos para acumuladores, por ejemplo, sino que también mejora la claridad del código al combinar la asignación con la declaración, haciendo el proceso más conveniente y semánticamente significativo.

¿Pero qué ocurre con los arrays? Normalmente un array contiene muchos más elementos de los que una inicialización simple puede cubrir cómodamente, por lo que el enfoque correcto depende mucho más del problema que con las variables simples. Por otro lado, es precisamente con los arrays donde la inicialización rápida resulta más útil. Imagina que estás

depurando un programa: en cada ejecución tendrías que teclear a mano una larga lista de valores — muy probablemente los mismos que en la ejecución anterior. Podrías copiarlos y pegarlos, pero ¿es eso siempre posible o conveniente? Rápidamente se vuelve tedioso y no ayuda en absoluto cuando estás buscando un error.

101: Inicialización rápida de arrays



TVec{42,-7,8,15,0,99,23,-3}

¿Cómo hace C++ la inicialización rápida con arrays? ¿Qué crees que mostrará el siguiente programa?

```C++
#include <iostream>
#include <array>
using namespace std;

const int N = 5;
typedef array<int,N> TVec;

int main()
{
    void printVec(TVec v);

    TVec garb;
    printVec(garb); // random numbers

    TVec v =       {1,2,3};
    printVec(v); // 1 2 3 0 0

    printVec(TVec{0,1,2,3,4});
    //            0 1 2 3 4

    return 0;
}

void printVec(TVec v)
{
  for (int i = 0; i < N; ++i)
      cout << v[i] << " ";
  cout << endl;
}
```

La respuesta es que, siendo un array un *agregado*, si se suministran menos valores de los que el array puede contener, el estándar dice que cada elemento restante se inicializa por copia desde una lista vacía `{}` — es decir, se inicializa por valor. Para `int`, la inicialización por valor significa inicialización a cero, por lo que esos elementos se convierten en `0`. En resumen: los valores que se escriben se asignan en orden, y el resto se pone a `0`.

Pero, ¡cuidado!, si no se da ningún valor inicial, el contenido del array es, como habitualmente con las variables simples, basura, memoria aleatoria dejada por otros programas.

Así que para inicializar un vector largo con todos los ceros, es suficiente:

```
TVec v = {};  // all the elements to 0
TVec w = {0}; // if you want to be more explicit
```

La «lista vacía» `{}` se rellena con el **cero** del tipo base: `0` para un array de `int`, `0.0` para `float`, y para `char` el carácter nulo. Por ejemplo, `typedef array<char, N> TChars; TChars cs = {};` rellena cada elemento con "`0`", el carácter `NUL` (código `0`), **no** el espacio `' '` (código 32). Para los booleanos, en un array de booleanos, es `false`.

A menudo necesitamos enviar valores literales a subprogramas. Con tipos simples esto es tan fácil como escribir el literal mismo

(`someFunction(3)`), pero para enviar un array completo como literal debemos indicarle al compilador el tipo exacto de ese literal — es decir, debemos hacer una conversión:

```
printVec(TVec{0,1,2,3,4,5,6,7,8,9});
```

enviará el array literal `{0,1,2,3,4,5,6,7,8,9}` y antes de eso, para que el compilador lo acepte como un array del mismo tipo exacto que el parámetro formal en `printVec(TVec)` espera, tenemos que «convertirlo» a `TVec`. Podemos usar esto para hacer pruebas rápidas de nuestros subprogramas; por ejemplo, para probar una función `TVec bubbleSort(TVec);` solo necesitamos unas pocas líneas en `main()`:

```C++
int main()
{
  void printVec(TVec);
  TVec bubbleSort(TVec);

  printVec(bubbleSort(TVec{9, 5, 4, 3, 2, 7}));
  printVec(bubbleSort(TVec{2, 4, 5, 1, 3, 4}));
  return 0;
}
// ..
```

A modo de resumen, la inicialización rápida es una herramienta muy útil que usaremos extensamente. Puedes establecer los valores iniciales de un array (no demasiado largo) de una vez por todas, usando la siguiente sintaxis sencilla:

```
1 typedef array<float,3> TVector;
2
3 TVector v1 = {3, 5, -2};    // quick init
4 TVector v2 = {{1,2}};       // first 2 initialised *
5 TVector v3 = {0}; // only the first is 0, the rest default to 0 too
```

En la línea 4 usamos `{{llaves, dobles}}` que son la forma más general de inicializar un array: las llaves exteriores inicializan el propio objeto `std::array`, y las llaves interiores inicializan el array interno sin procesar que encapsula.

`{llaves simples}` también funcionan: el compilador infiere las llaves interiores faltantes (*elisión de llaves*). Para arrays de arrays o `struct`s, es más claro escribir todas las llaves.

Imagina probar un programa con arrays de `10` o `20` elementos: cada vez que lo ejecutas para probarlo, tendrías que teclear/pegar cada número, probablemente los mismos que la vez anterior. La inicialización rápida es recomendable para estos casos, y es un gran ahorro de tiempo durante la depuración.

```C++
#include <iostream>
#include <array>
using namespace std;

const int N = 3;
typedef array<int,N> TVec;

int main()
{
  int sum(TVec);

  TVec a = {1, 2, 3};

  cout << sum(a) << endl;
```

```
  return 0;
}

int sum(TVec v)
{
  int s = 0;
  for (int i = 0; i < N; ++i)
    s += v[i];
  return s;
}
```

4·2.4 Ejemplos con Arrays

La mejor manera de aprender a usar arrays es construyendo cosas con ellos. Dado un tipo como

```
const int N = 3;
typedef array<int,N> TVec;
```

una «caja de herramientas» típica de subprogramas a dominar sería:

```
void initArr(TVec& a, int value = 0);
void printArr(TVec a);
void readArr(TVec& a);
TVec readArr();
int  sumArr(TVec a);
bool isIn(TVec a, int x);
int  indexOf(TVec a, int x);
int  count(TVec a, int x);
bool anyReps(TVec a);
int  findPosFirstMax(TVec a);
```

Algunas de estas signaturas merecen un comentario:

- `initArr` y la primera forma de `readArr` **modifican** el array, por lo que lo reciben por referencia (`TVec&`); `printArr`, `sumArr`, `isIn`, `indexOf`, `count`, `anyReps` y `findPosFirstMax` solo **leen** el array, por lo que se pasa por valor.
- `readArr` se muestra en dos estilos distintos: uno rellena un parámetro por referencia y no devuelve nada, el otro construye un array nuevo y lo devuelve (`TVec readArr()`). Ambos son correctos e idiomáticos; elige un estilo y mantenlo consistente dentro del mismo programa.
- `indexOf` y `findPosFirstMax` devuelven una posición como `int`; por convención, un valor como `-1` se usa para indicar «no encontrado».

Como ejemplo resuelto, aquí están `initArr`, que rellena cada celda con un valor dado, y `printArr`, que imprime los elementos en orden, separados por espacios:

```
void initArr(TVec& a, int value = 0)                    C++
{
  for (int i = 0; i < N; ++i)
    a[i] = value;
}

void printArr(TVec a)
{
  for (int i = 0; i < N; ++i)
    cout << a[i] << " ";
  cout << endl;
}
```

Un segundo ejemplo merece estudiarse con detenimiento, porque ilustra la regla `for`/`while`. `isIn` nos dice si un valor `x` aparece en el array. Como podemos detenernos en el momento en que lo encontramos, el número

de iteraciones **no** se conoce de antemano — así que el bucle de elección es `while`, no `for`:

```C++
bool isIn(TVec a, int x)
{
  int i = 0;
  bool found = false;
  while (i < N and not found) {
    found = (a[i] == x);
    ++i;
  }
  return found;
}
```

Nótese el estilo disciplinado: un único `return`, sin `break` para salir del bucle, y los operadores en palabras `and`/`not`. Compara esto con `sumArr`, que debe **siempre** visitar cada celda para sumarla: allí el número de iteraciones es fijo en `N`, así que `for` es la elección correcta. Elegir entre `for` y `while` no es cuestión de gusto — lo dicta el hecho de si el número de repeticiones se conoce antes de que comience el bucle.

Intenta implementar el resto — `readArr`, `sumArr`, `isIn`, `indexOf`, `count`, `anyReps` y `findPosFirstMax` — por tu cuenta antes de pasar a los ejercicios.

Entrada/salida de arrays: Como habrás notado, no hemos hablado de `cin >> a` (ni de `cout << a`) porque estas operaciones no existen. Como los arrays son tipos definidos por el programador, `cin`/`cout` no tiene ningún mecanismo integrado para manejarlos. Es por tanto responsabilidad del programador implementar rutinas personalizadas para leer y escribir arrays.

4·2.5 Punto de control

1) ¿Cuál sería el contenido del array `cs` tras `typedef array<char, 3> TChars; TChars cs = {'a'};`?
2) Dado `typedef array<int, 5> TVec; TVec v = {7, 8};`, ¿qué contiene `v`? ¿Y qué ocurre con un `TVec w;` declarado sin inicializador?
3) Para un `array<int, 10> a;`, ¿cuál es el rango válido de índices? A nivel de memoria, ¿qué ocurre cuando se escribe en `a[10]`?
4) ¿Cuál es la diferencia entre `a[i]` y `a.at(i)`? ¿Cuándo preferirías cada uno?
5) ¿Por qué debe darse nombre a un tipo array con `typedef` antes de que pueda pasarse a un subprograma? ¿Qué ocurre si dos arrays estructuralmente idénticos se declaran sin compartir un `typedef`?
6) `.size()` devuelve el número de elementos de un array. ¿Por qué podría documentar mejor una condición de bucle una constante con nombre como `N` que `a.size()`, aunque ambas sean correctas?
7) Explica la diferencia entre los inicializadores `{{3, 5, -2}}` y `{3, 5, -2}`. ¿Por qué existen las llaves dobles?
8) ¿Cómo inicializarías un `array<float, 100>` para que todos sus elementos sean `0.0`, escribiendo lo mínimo posible?

9) Dos arrays del mismo tipo se comparan con `a < b`. Describe, con tus propias palabras, la regla usada para decidir el resultado. ¿A qué ordenación cotidiana se parece?

10) Al recorrer un array, ¿cuándo debes usar `for` y cuándo `while`? Da un ejemplo de una línea de una tarea que necesite cada uno.

11) ¿Por qué `cin >> a` y `cout << a` no funcionan para un array `a`? ¿De quién es la responsabilidad, entonces, de leer e imprimir arrays?

12) De las funciones de la caja de herramientas, ¿cuáles reciben su array por referencia (`TVec&`) y cuáles por valor (`TVec`)? ¿Cuál es el criterio que lo decide?

13) Escribe el cuerpo de `int sumArr(TVec a)` y de `bool anyReps(TVec a)` (true si algún valor aparece más de una vez). Para cada uno, decide si `for` o `while` es apropiado y justifica tu elección.

Tarea 9: Arrays

A lo largo de estos ejercicios, asume la siguiente configuración:

```cpp
#include <array>
const int N = 3;
typedef array<int, N> TVec;
```

aunque los tipos y tamaños específicos pueden variar por ejercicio. Usamos *posición* como sinónimo de *índice*: la primera posición es `0` y la última es `N-1` (equivalentemente, `a.size()-1`).

Para evitar reescribir los datos de prueba en cada ejecución, inicializamos los arrays con literales inicializados entre llaves directamente en `main()` y los pasamos a subprogramas como argumentos. El ejercicio 1 es la excepción: requiere leer y escribir desde y hacia el terminal.

4·2.6 Una trampa: los tamaños son sin signo

Los valores devueltos por `size()` y `length()` no son `int`: son números *sin signo*, que no pueden ser negativos. Es una herencia histórica de C, y provoca una sorpresa desagradable cuando un `unsigned` y un `int` se encuentran en la misma expresión: el `int` se convierte silenciosamente a `unsigned`, y un valor negativo se convierte en un número positivo enorme.

```cpp
string s = "ab";
cout << s.length() - 3 << endl;   // imprime 18446744073709551615, no -1

int i = -1;
if (i < s.length())               // ¡falso!  -1 se convierte en un número enorme
    cout << "-1 < 2" << endl;
```

Desde C++20 también existe una función libre `ssize(s)` que devuelve un valor con signo, pero seguiremos usando `int(s.length())`, que funciona en todas las versiones del lenguaje.

Nuestra regla: convierte el tamaño a `int` en cuanto lo obtengas, para que los valores `unsigned` nunca participen en tu aritmética ni en tus comparaciones. Escribe la conversión explícitamente:

```cpp
for (int i = 0; i < int(s.length()); ++i)
```

o guarda el tamaño en un `int` una vez y usa esa variable:

```cpp
int n = s.length();
```

Con esta regla, `int(s.length()) - 3` es `-1` y `i < int(s.length())` es `true`, como es de esperar. Algunos compiladores pueden advertir sobre estas situaciones, pero los buenos hábitos no deben depender de las advertencias.

1 Escribe un programa `main()` que declare los siguientes subprogramas, pruebe cada uno con argumentos `TVec` apropiados (usando literales inicializados entre llaves donde sea posible) e imprima los valores devueltos: › Sol

```cpp
void initArr0(TVec& a);       // set every element to 0
void printArr(TVec a);        // print all elements separated by spaces
void readArr(TVec& a);        // read N values from the keyboard (procedure)
```

```
TVec readArr();               // read N values from the keyboard (function)
int  sumArr(TVec a);          // return the sum of all elements
bool isIn(TVec a, int x);     // return true if x appears in a
int  indexOf(TVec a, int x);  // return the index of the first occurrence of x
int  count(TVec a, int x);    // count how many times x appears
bool anyReps(TVec a);         // return true if any element is repeated
int  findPosFirstMax(TVec a); // return the index of the (first) maximum
```

2 Usando `TVec` del ejercicio 1, construye un procedimiento `void printRev(TVec a)` que imprima el array en orden inverso. Por ejemplo: › Sol

```
int main()                                                    C++
{
    void printRev(TVec a);

    printRev(TVec{{1, 20, 3, 40, 5}}); // 5 40 3 20 1

    cout << endl; // necessary to flush data on the screen
    return 0;
}
```

También puedes probarlo con `printRev(readArr())`: leer un array y pasarlo inmediatamente a `printRev()`.

3 Define un nuevo tipo `THeights` como un array de `N = 5` floats. Declara una variable `alumHeights`, inicialízala con cinco valores (p.ej. `1.83 1.78 1.55 1.90 1.60`), y construye la función `float mean(THeights a)` que devuelva su media. Comprueba que el ejemplo da `1.732`. › Sol

4 Usando `TVec` con elementos `float`, construye la función `float scalarProd(TVec a, TVec b)` que devuelva el producto escalar (dot) de dos vectores. El producto escalar es $\sum_{i=0}^{N-1} a_i \cdot b_i$. › Sol

```
int main()                                                    C++
{
    void printArr(TVec a);
    float scalarProd(TVec a, TVec b);

    TVec a = {{1, 2, 3}},
         b = {{3, 1, 1}};
    printArr(a); cout << endl; // 1 2 3
    printArr(b); cout << endl; // 3 1 1
    cout << "Scalar product: "
         << scalarProd(a, b)   // 8
         << endl;
    return 0;
}
```

5 Construye una función `int numberOfNeg(TVec a)` que devuelva el número de elementos negativos en `a`. › Sol

```
int main()                                                    C++
{
    void printArr(TVec a);
    int numberOfNeg(TVec a);

    TVec a = {{1, -22, 0}};
    printArr(a); cout << endl;
    cout << "Negatives: " << numberOfNeg(a) << endl; // 1
```

```
    a = TVec{{0, 1, 0}};
    printArr(a); cout << endl;
    cout << "Negatives: " << numberOfNeg(a) << endl; // 0

    return 0;
}
```

6 Construye una función `float minVec(TVec a)` que devuelva el valor más pequeño en `a`. › Sol

```
int main()                                                    C++
{
    void printArr(TVec a);
    float minVec(TVec a);

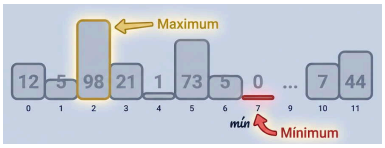
    TVec a = {{1, -2, 10}};
    printArr(a); cout << endl;
    cout << "Min: " << minVec(a) << endl; // -2

    a = TVec{{-10, -1, -20}};
    printArr(a); cout << endl;
    cout << "Min: " << minVec(a) << endl; // -20

    a = TVec{{4, -10, -2}};
    printArr(a); cout << endl;
    cout << "Min: " << minVec(a) << endl; // -10

    return 0;
}
```

7 Construye un procedimiento `void minMaxVec(TVec a, int& mn, int& mx)` que devuelva tanto el mínimo como el máximo de `a` a través de parámetros por referencia. › Sol



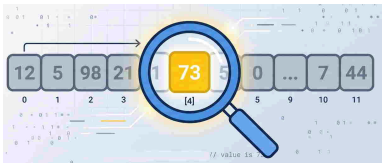

```
int main()                                                    C++
{
    void printArr(TVec a);
    void minMaxVec(TVec a, int& mn, int& mx);

    int mn, mx;
    TVec a = {{1, -22, 0}};
    printArr(a); cout << endl;
    minMaxVec(a, mn, mx);
    cout << "Min: " << mn << endl; // -22
    cout << "Max: " << mx << endl; // 1

    a = TVec{{0, 1, 0}};
    printArr(a); cout << endl;
    minMaxVec(a, mn, mx);
    cout << "Min: " << mn << endl; // 0
    cout << "Max: " << mx << endl; // 1

    return 0;
}
```

8 **(Buscar.)** Construye una función `int findArr(TVec a, int x)` que devuelva la **primera posición** en la que `x` aparece en `a`, o -1 si no se encuentra. › Sol




```
int main()                                                    C++
{
    void printArr(TVec a);
    int findArr(TVec a, int x);

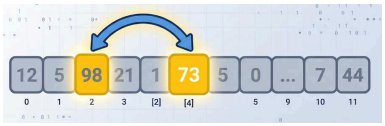
    TVec a = {{1, -22, 0}};
    printArr(a); cout << endl;
    cout << "find 1: "  << findArr(a,  1) << endl; // 0
```

```
    a = TVec{{0, 2, 10}};
    printArr(a); cout << endl;
    cout << "find 10: " << findArr(a, 10) << endl; // 2

    return 0;
}
```

9 Usando `void swap(int& a, int& b)` — que intercambia el **contenido** de `a` y `b` — construye un procedimiento `void invertArr(TVec& a)` que invierta los elementos de `a` en su lugar.

HINT: Solo hay que intercambiar la primera mitad del array con la segunda mitad; ten cuidado de no intercambiar elementos dos veces. › Sol




```
int main()                                                    C++
{
    void printArr(TVec a);
    void invertArr(TVec& a);

    TVec a = {{1, 2, 3, 4, 5}};
    printArr(a); cout << endl;
    invertArr(a);
    cout << "after invert:" << endl;
    printArr(a); cout << endl; // 5 4 3 2 1

    return 0;
}
```

10 Construye una función `int find12(TVec a)` que devuelva el primer índice en el que el par `1, 2` aparece consecutivamente en `a`, o `−1` si no aparece. Por ejemplo, en `{-2, 3, 1, 2, 0, 1, 2}` devuelve `2`.

HINT: Ten cuidado de no leer más allá del último elemento del array. › Sol

```
int main()                                                    C++
{
    void printArr(TVec a);
    int find12(TVec a);

    TVec a = {{3, 1, 2, 3, 4}};
    printArr(a); cout << endl;
    cout << "1,2 at: " << find12(a) << endl; // 1

    a = TVec{{3, 2, 2, 2, 1}};
    printArr(a); cout << endl;
    cout << "1,2 at: " << find12(a) << endl; // -1

    return 0;
}
```

11 Representa un punto 3-D como `typedef array<float, 3> TPos`. Dados dos puntos `r1` y `r2`, construye una función `float distance(TPos r1, TPos r2)` que devuelva la distancia euclídea entre ellos. Necesitarás `sqrt()` de `<cmath>`. › Sol

```
int main()                                                    C++
{
    float distance(TPos r1, TPos r2);

    TPos a = {{0, 0, 1}},
         b = {{0, 1, 0}};
    cout << "Distance: " << distance(a, b) << endl; // 1.41421
```

```
    return 0;
}
```

12 Usando un `TVec` de `N` floats, construye:

```
void stats(TVec a, float& mean, float& mn, float& mx, float& stdev);
```

que calcule la media, el mínimo, el máximo y la desviación típica muestral corregida de `a`:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{N-1} (a_i - \overline{a})^2}{N-1}}$$

donde $\overline{a}$ es la media. Esa fórmula requiere dos pasadas sobre el array (una para la media, otra para las diferencias al cuadrado). Intenta implementarlo así primero; después prueba la fórmula algebraicamente equivalente de una sola pasada:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{N-1} a_i^2 - \left(\sum_{i=0}^{N-1} a_i\right)^2 / N}{N-1}}$$

› Sol

```
int main()                                                    C++
{
    void printArr(TVec a);
    void stats(TVec a, float& mean, float& mn,
               float& mx, float& stdev);

    float mean, mn, mx, stdev;
    TVec a = {{1, 2, 3, 4, 5}};
    printArr(a); cout << endl;
    stats(a, mean, mn, mx, stdev);
    cout << "mean:    " << mean  << endl; // 3
    cout << "min:     " << mn    << endl; // 1
    cout << "max:     " << mx    << endl; // 5
    cout << "std dev: " << stdev << endl; // 1.58114

    return 0;
}
```

13 Usando el procedimiento del ejercicio 12, calcula la media, el mínimo, el máximo y σ de las temperaturas medias anuales en Málaga de 1906 a 2025 (datos de weatherandclimate.eu, columna «año»). Establece `N = 120` y usa los datos siguientes. Después llama a `fitLine()` (proporcionada) para ajustar una línea recta $y = ax + b$ a los datos, donde $x = 1$ es el año 1906, e indica la pendiente, la ordenada en el origen y el error RMS. › Sol

```
int main()                                                    C++
{
    void stats(TVec a, float& mean, float& mn,
               float& mx, float& stdev);
    void fitLine(TVec y, float& a, float& b, float& rms);
    void printArr(TVec a);

    TVec means = {{
        16.5, 17.1, 17.7, 17.0, 17.1, 17.4, 17.6, 17.3, 17.5, 17.8,
        17.4, 17.1, 17.3, 17.3, 18.1, 17.5, 17.6, 16.9, 17.7, 17.2,
```

```
        18.0, 18.2, 17.6, 18.0, 17.8, 18.2, 17.2, 17.7, 17.2, 17.5,
        17.3, 18.0, 17.4, 17.4, 17.7, 17.3, 17.3, 17.5, 17.2, 18.1,
        17.5, 18.2, 18.1, 18.6, 18.4, 17.7, 18.3, 17.9, 17.5, 18.7,
        16.8, 17.5, 18.0, 18.0, 17.8, 17.9, 18.2, 17.9, 18.0, 18.0,
        18.2, 17.7, 17.8, 17.3, 17.6, 17.1, 17.0, 17.2, 17.4, 17.4,
        17.3, 17.7, 17.8, 17.8, 17.8, 18.0, 17.8, 18.3, 17.6, 18.1,
        18.0, 18.5, 18.3, 18.6, 18.6, 17.8, 17.9, 17.6, 18.5, 19.2,
        18.0, 18.9, 18.5, 18.6, 18.3, 19.3, 19.2, 19.0, 18.7, 18.6,
        19.0, 19.0, 19.0, 19.2, 18.9, 19.3, 19.0, 19.1, 19.9, 19.7,
        19.6, 19.3, 19.0, 19.6, 19.5, 19.3, 20.0, 20.1, 20.0, 19.8
    }};
    printArr(means); cout << endl;

    float mean, mn, mx, stdev;
    stats(means, mean, mn, mx, stdev);
    cout << "mean:    " << mean  << endl; // 18.0833
    cout << "min:     " << mn    << endl; // 16.5
    cout << "max:     " << mx    << endl; // 20.1
    cout << "std dev: " << stdev << endl; // 0.799303 (changes with
    method)

    float a, b, rms;
    fitLine(means, a, b, rms);
    cout << "a = " << a << " degC/year" << endl; // 0.0179966
    cout << "b = " << b << " degC"      << endl; // 16.9945
    cout << "RMS = " << rms << " degC"  << endl; // 0.495051

    return 0;
}
```

Usa el siguiente auxiliar para ajustar la línea:

```
void fitLine(TVec y, float& a, float& b, float& rms)                C++
{
    float sx = 0, sy = 0, sxy = 0, sx2 = 0;
    for (int i = 0; i < N; ++i) {
        float x = i + 1;
        sx  += x;
        sy  += y[i];
        sxy += x * y[i];
        sx2 += x * x;
    }
    float meanx = sx / N, meany = sy / N;
    a = (sxy - sx * meany) / (sx2 - sx * meanx);
    b = meany - a * meanx;

    float sumsq = 0;
    for (int i = 0; i < N; ++i) {
        float x = i + 1;
        float diff = y[i] - (a * x + b);
        sumsq += diff * diff;
    }
    rms = sqrt(sumsq / N);
}
```

La pendiente te indica cuántos grados por año se está calentando Málaga en esta fecha.

14 Construye una función `int sum2Biggest(TVec a)` que devuelva la suma de los dos elementos más grandes de `a`. › Sol

```
int main()                                                          C++
{
    int sum2Biggest(TVec a);

    cout << sum2Biggest(TVec{{1, 2, 3, 4, 5}}) << endl; //  9
    cout << sum2Biggest(TVec{{1, 5, 3, 4, 5}}) << endl; // 10
```

```
    return 0;
}
```

o O o

Tarea 10: Arrays++

A lo largo de estos ejercicios, asume la siguiente configuración:

```cpp
#include <array>
const int N = 5;  // for example
typedef array<int, N> TVec;
```

aunque los tipos y tamaños específicos pueden variar por ejercicio. Usamos *posición* como sinónimo de *índice*: la primera posición es `0` y la última es `N-1` (equivalentemente, `a.size()-1`). Para muchos problemas es útil tener ya implementado `void printArr(TVec a)`. La manera más sencilla de probar desde `main()` es pasar arrays inicializados con llaves directamente como argumentos.

1 Construye un subprograma que reciba un `TVec` y un valor `x`, y devuelva la **última** posición en la que aparece `x`. Si el valor no se encuentra, devuelve -1: › Sol

```cpp
int rfind(TVec a, int x);
```

Por ejemplo, `rfind(TVec{{3,15,9,0,9}}, 9)` debería devolver `4`.

2 Supón que `TVec` puede contener elementos repetidos. Define la **distancia** entre dos elementos iguales como la diferencia entre sus posiciones. Por ejemplo, en `3 4 3 5` los dos `3`s están a distancia 2. Construye un subprograma que encuentre la **mayor distancia entre repeticiones** de su segundo parámetro. Devuelve -1 si el valor está ausente; devuelve 0 si aparece solo una vez: › Sol

```cpp
int maxDistBetwReps(TVec a, int x);
int main()
{
    int maxDistBetwReps(TVec a, int x);

    cout << maxDistBetwReps(TVec{{10,10,4,3,5,6}}, 10) << endl; //  1
    cout << maxDistBetwReps(TVec{{ 1, 3,4,3,5,3}},  3) << endl; //  4
    cout << maxDistBetwReps(TVec{{ 1, 3,4,3,5,6}},  1) << endl; //  0
    cout << maxDistBetwReps(TVec{{ 1, 3,4,3,5,6}}, 33) << endl; // -1

    return 0;
}
```

3 Construye un subprograma que devuelva la primera posición en la que aparece un elemento repetido. Si ningún elemento está repetido, devuelve -1: › Sol

```cpp
int firstRep(TVec a);
```

Por ejemplo:

- `firstRep(TVec{{1, 2, 3, 2}})` ⟶ `1`
- `firstRep(TVec{{1, 2, 1, 2}})` ⟶ `0`
- `firstRep(TVec{{1, 2, 3, 4}})` ⟶ `-1`

4 Construye un subprograma que encuentre el **primer pico** de un array. Un pico es un elemento estrictamente mayor que ambos vecinos ($a_{i-1} < a_i > a_{i+1}$). El primer y último elemento son picos si son estrictamente mayores que su único vecino. Devuelve la posición del pico, o -1 si no existe ninguno: › Sol

```cpp
int firstPeak(TVec a);
```

- `firstPeak(TVec{{3,2,9,7}})` ⟶ 0
- `firstPeak(TVec{{3,3,3,3}})` ⟶ −1
- `firstPeak(TVec{{3,3,3,4}})` ⟶ 3
- `firstPeak(TVec{{3,3,5,4}})` ⟶ 2

5 🧊 Determina si dos arrays son **circularmente iguales**: uno es una rotación del otro. Supón que ningún array contiene elementos duplicados. › Sol

```
bool circularlyEq(TVec a, TVec b);
```

Por ejemplo,
- `{{1,2,3,4,5}}` y `{{5,1,2,3,4}}` son circularmente iguales, mientras que
- `{{1,2,3,4,5}}` y `{{1,2,3,5,4}}` no lo son.

6 🧊 Cambia `N` a 15 y construye un subprograma que devuelva la longitud de la racha más larga de elementos iguales consecutivos: › Sol

```
int lcs(TVec a);
```

Una *racha* es una subsecuencia maximal de elementos idénticos.

`lcs(TVec{{3,9,9,0,9,3,9,9,9,0,9,3,9,9,0,9}})` ⟶ 3.

Si ningún elemento está repetido, devuelve 1.

7 🧊 Construye un subprograma que encuentre la **tendencia** de un array: el elemento que aparece estrictamente con más frecuencia que cualquier otro. Si dos o más elementos empatan en la frecuencia más alta, no hay tendencia:

```
void trend(TVec a, int& tr, bool& ok);
```

Usa `ok` para indicar si existe una tendencia. › Sol

```
2 5 5 2 5 2 5 5  trend: 5                                   example
1 1 1 1 1 1 1 1  trend: 1
2 1 2 2 1 1 2 1  has NO trend
```

8 Supón que se dan las coordenadas en punto flotante (x_i, y_i) de una nube de puntos, y queremos encontrar la recta de mejor ajuste $y = ax + b$ — la recta más cercana a todos los puntos en el sentido de los mínimos cuadrados. La pendiente y la ordenada en el origen vienen dadas por: › Sol

$$a = \frac{n\sum_{i=0}^{n-1} x_i y_i - \sum_{i=0}^{n-1} x_i \sum_{i=0}^{n-1} y_i}{n\sum_{i=0}^{n-1} x_i^2 - \left(\sum_{i=0}^{n-1} x_i\right)^2}$$

$$b = \frac{\sum_{i=0}^{n-1} y_i - a\sum_{i=0}^{n-1} x_i}{n}$$

Como b depende de a, primero hay que calcular a. La función devuelve dos resultados, a y b, por lo que la interfaz usa parámetros por referencia:

```
void linearApproach(TVec x, TVec y, float& a, float& b);
```

Aquí `TVec` es un array de `float`. Llámala desde `main()` con:

```
linearApproach({{0,1,3,5,7}}, {{1,0,2,4,6}}, a, b);
```

y verifica que $a \approx 0.804878$ y $b \approx 0.024390$.

9 **Purgar.** En informática, *purgar* significa eliminar duplicados de una lista. El antiguo sistema operativo VAX/VMS ofrecía un comando `PURGE` que borraba versiones antiguas de ficheros (que se numeraban con un sufijo como `name.ext;122`). Construye tanto un procedimiento como una función que purguen un `TVec` conservando solo la **primera** aparición de cada elemento y eliminando cualquier duplicado posterior: › Sol

```
void purge(TVec& v);
TVec purge(TVec v);
```

La versión procedural debe funcionar en su lugar, sin copiar el array ni usar almacenamiento extra. Puede ser útil implementar primero:

```
void deletePos(TVec& v, int pos);
```

Dado que el array resultante es más corto que el original, rellena las posiciones finales no usadas con ceros. (Esto funciona siempre que el array no contenga legítimamente ceros.)

10 **Lanzar Dardos.** Escribe una función que acepte un array de radios de lanzamiento (`N = 3` enteros) y devuelva una puntuación total según estas reglas: › Sol

- `0` puntos — radio `> 10`
- `5` puntos — `5 <= radio <= 10`
- `10` puntos — radio `< 5`
- BONUS: si todos los radios son `< 5`, suma `100` puntos extra

```
scoreThrows(TVec{{ 1,  5, 11}}) // returns  15          example
scoreThrows(TVec{{15, 20, 30}}) // returns   0
scoreThrows(TVec{{ 1,  2,  3}}) // returns 130
```

11 Escribe una función que acepte un array y un entero `n`, y devuelva el número de pares de índices (i, j) con $i < j$ tales que $|a_i - a_j| = n$: › Sol

```
int pairsDiff(TVec a, int n);
pairsDiff(TVec{{1,1,5,6,9}}, 4);
    // 3: pairs (0,2), (1,2), (2,4)
pairsDiff(TVec{{1,1,3,3,9}}, 2);
    // 4: pairs (0,2), (0,3), (1,2), (1,3)
```

12 **Suma más cercana.** Dado un array desordenado de enteros y un valor objetivo, encuentra el par de elementos cuya suma sea más cercana al objetivo. Devuelve los dos valores a través de parámetros por referencia: › Sol

```
void closestSum(TVec ns, int goal, int& a, int& b);
```

(Este es un problema clásico de *Programming Pearls* de Jon Bentley.) Escribe también un auxiliar `void test(TVec a, int goal)` que imprima:

```
goal:  4;  array:  9 -3 -10 -4 -5   ->  9 + -5          Display
goal:  0;  array:  1 -1  -2 -4  0   ->  1 + -1
goal:  6;  array:  2  2 -10 10  5   ->  2 +  5
```

cuando se llame desde `main()` con:

```
test(TVec{{ 9, -3, -10, -4, -5}}, 4);                   C++
test(TVec{{ 1, -1,  -2, -4,  0}}, 0);
test(TVec{{ 2,  2, -10, 10,  5}}, 6);
```

13 **Suma del punto medio.** Dado un array de enteros, devuelve el índice del elemento para el cual la suma de todos los elementos a su izquierda es igual a la suma de todos los elementos a su derecha. Devuelve -1 si no existe tal elemento: › Sol

```
int midpointSum(TVec a);
```

HINT: El elemento en el índice devuelto no se incluye en ninguna de las sumas. El primer y último índice no pueden ser puntos medios.

```
midpointSum(TVec{{ 4, 1, 7, 9,  3, 9}}) // 3  (4+1+7=12, 3+9=12)   C++
midpointSum(TVec{{ 4, 4, 4, 4,  4, 4}}) // -1
midpointSum(TVec{{16, 4, 4, 4,  4, 4}}) // 1
```

14 **Distancia Manhattan.** La distancia Manhattan entre dos puntos de una cuadrícula es el número total de pasos horizontales y verticales necesarios para ir de uno al otro (nombre tomado del trazado en cuadrícula de las calles de Manhattan, Nueva York). Escribe: › Sol

```
int manhattanDistance(TCoord a, TCoord b);
```

donde `typedef array<int,2> TCoord` almacena coordenadas (x, y).

```
manhattanDistance([1,1], [1,1]) // 0                    example
manhattanDistance([5,4], [3,2]) // 4
manhattanDistance([1,1], [0,3]) // 3
```

15 **Valor atípico de paridad.** Un array contiene enteros que son todos impares o todos pares, excepto exactamente uno diferente. Devuelve la **posición** de ese valor atípico. Si más de un elemento tiene una paridad distinta a la del resto, devuelve -1: › Sol

```
int parityOutlier(TVec a);
```

```
parityOutlier({{  0, 100,  4, 11}}) // [3] -> 11        example
parityOutlier({{160,  19, 11, 13}}) // [0] -> 160
parityOutlier({{  1,   1,  2,  1}}) // [2] -> 2
parityOutlier({{  1,   1,  2,  2}}) // -1  (no single outlier)
```

16 Rellena un array de `100` floats con valores aleatorios usando: X

```
void randFloatArr(TVec& a, float min, float max)        C++
{
    for (int i = 0; i < int(a.size()); ++i)
        a[i] = (float(rand()) / RAND_MAX) * (max - min) + min;
}
```

Después imprime el array y su media.

17 Usando `randFloatArr()` del ejercicio anterior, rellena un array aleatorio, imprímelo, ordénalo con el procedimiento siguiente y muestra el resultado ordenado: X

```cpp
void bubbleSort(TVec& a)
{
    for (int top = int(a.size()) - 1; top >= 1; --top)
        for (int i = 0; i < top; ++i)
            if (a[i] > a[i + 1])
                swap(a[i], a[i + 1]);
}
```

o O o

4·3 Cadenas de caracteres

Las cadenas de caracteres permiten a los ordenadores procesar texto y comunicarse con los usuarios

En su forma escrita o hablada, el texto sigue siendo el único medio fiable para transmitir conceptos abstractos. En consecuencia, continúa siendo la base principal de los sistemas de información, a pesar de los esfuerzos sustanciales por incorporar medios alternativos como interfaces gráficas, sistemas de realidad virtual y síntesis animada. Este papel fundamental se ve reforzado por la proliferación de bases de datos de conocimiento —jurídicas, comerciales y de otro tipo— desplegadas en Internet, especialmente a través de servicios web y, más recientemente, por la aparición de los Modelos de Lenguaje Grandes (LLM), cuyo medio de trabajo natural es el texto. El texto sigue siendo mucho más eficiente que las imágenes u otros medios para estos fines. Las imágenes, salvo que estén etiquetadas y convertidas en cadenas de etiquetas, no son adecuadas para el razonamiento o la precisión lógica. Las matemáticas se basan en representaciones textuales de símbolos y expresiones; el lenguaje natural es inherentemente textual; y la programación, a cualquier nivel —desde expresiones de alto nivel hasta código máquina— es fundamentalmente texto.

El texto en C++ se gestiona mediante `char`s codificados y `string`s de chars.

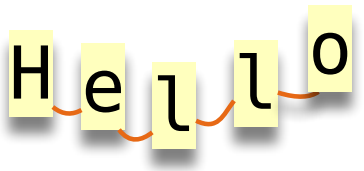
Recuerde (Sección 2·4) que un único carácter se representa con comillas simples:

```
char c = 'X';
```

Sin embargo, para representar una palabra, una frase, un párrafo o cualquier fragmento de texto más largo, necesitamos más de un carácter — necesitamos un array de caracteres. Aunque algo como `array<char, 256> aVarString;` es posible, las cadenas de caracteres son tan fundamentales para la programación que C++ les otorga su propio tipo dedicado, `string`, en lugar de obligarnos a gestionar manualmente un `array<char, N>` en bruto. En comparación con un array simple de `char`, un `string`:

- No tiene límite de tamaño fijo — un `string` crece y decrece dinámicamente, a diferencia de un `array`, cuyo tamaño `N` queda fijado para siempre en el momento de la declaración.
- Viene con operaciones especializadas de entrada/salida, con variantes como `cin >> s` y `getline(cin, s)` que leen una sola palabra o una línea completa de una vez.
- Ofrece toda una familia de operaciones especializadas (búsqueda, sustitución, inserción, comparación...) optimizadas para trabajar con texto.

105: Las cadenas de caracteres son fundamentales para el procesamiento de información y están presentes en todos los aspectos de la programación



4·3.1 Definición. El tipo de datos string

Declarar una cadena de caracteres no puede ser más sencillo:

```
string name;
string address = "C/ Larios 6, Malaga";
name = "Juan";
```

En rigor, `string` vive en la cabecera `<string>` y debería incluirse explícitamente con `#include`. En la práctica, si ya se ha incluido `<iostream>`, `string` funcionará igualmente en la mayoría de los casos, porque `<iostream>` arrastra `<string>` de forma indirecta en la mayoría de las implementaciones — pero esto no está garantizado por el estándar.

No olvide que los literales de cadena siempre se escriben entre comillas dobles, y que es perfectamente válido tener una cadena vacía, `s = "";`. De hecho, ni siquiera es necesario preocuparse por inicializar explícitamente un string a vacío: por definición, un `string` recién declarado ya comienza vacío, así que `string name;` por sí solo garantiza que `name` es `""`.

106: La declaración de un string es sencilla, siendo su valor inicial cadena vacía. Para leerlos se usan `cin` y `getline()`

```
string s;
string name = "Juan";
getline(cin, s);
if (name.length() > s.size()) {
```

4·3.2 Operaciones con cadenas de caracteres

A diferencia de un `array`, cuyo tamaño `N` queda fijado para siempre, un `string` **sí puede** crecer y decrecer — esa es precisamente su principal ventaja sobre un array simple de `char`. Lo que no se puede hacer es crecer o decrecer **mediante indexación**: `s[i]` solo permite leer o sobreescribir un carácter que ya existe dentro de la longitud actual del string; nunca puede usarse para añadir un carácter más allá del final, ni para eliminar uno. Para cambiar la longitud de un string hay que usar alguna de sus propias operaciones — como `+=`, `.append()`, `.insert()`, `.erase()` — o reasignarlo completamente. Por ejemplo:

```
string s = "cloud";
s += 's'; // now s contains "clouds" -- its length grew by one
```

Se puede sobreescribir libremente una posición existente, `s[0] = 'C'`, pero no se puede usar la indexación para añadir un carácter más allá del final actual del string: ~~`s[5] = 's'`~~ es comportamiento indefinido si `s` solo tiene 5 caracteres (posiciones `0`..`4`). Lo mismo ocurre a la inversa: la indexación por sí sola nunca puede reducir la longitud de un string. Crecer y decrecer solo es posible mediante las propias operaciones del string, como las mostradas anteriormente.

El método más importante (en las clases, los subprogramas internos se denominan **métodos**) para los strings es `.length()` — o, de forma equivalente, `.size()`, ya que `string`, como los demás tipos de contenedor STL, también soporta esta interfaz común — que nos indica la propiedad global más importante de un string: su número de caracteres. Así:

```
int main()                                                    C++
{
  string s; // s.length() -> 0
}
```

Para construir un nuevo string a partir de uno existente, se añaden caracteres de uno en uno y se reasigna el resultado a una variable (posiblemente nueva):

```
string toLowercase(string s)                                  C++
{
  string r;
  for (int i = 0; i < int(s.size()); ++i)
    r += tolower(s[i]);
    // r.append(1,tolower(s[i])); // an equivalent alternative
  return r;
}
```

Tenga en cuenta que `s.size()` devuelve un tipo sin signo; por eso lo envolvemos en `int(...)` en el bucle anterior, como se explicó en «Una trampa: los tamaños son sin signo».

Las operaciones de **entrada/salida** sobre strings se presentan en dos modalidades:

```
string s;
cin >> s;              // reads only the next word (stops at the first
blank or newline)
cout << s << endl;

getline(cin, s);       // reads a whole line, until \n
getline(cin, s, '\t'); // reads until a tab character
```

Existe un error clásico cuando se mezclan ambas. `cin >> s` se detiene en cuanto encuentra el primer carácter de espacio en blanco — incluido el salto de línea producido al pulsar Enter — pero deja ese salto de línea sin leer en el buffer de entrada. Si una llamada a `getline` viene justo después, consumirá el resto de esa **misma** línea — lo que quede tras la palabra, que puede estar incluso vacío — en lugar de esperar una nueva línea de entrada:

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

int main()
{
  string s;
  cin >> s;            // does not read "spaces"
  cout << s << endl;
  getline(cin, s);     // reads only what is left of the
                       // SAME line, not a new one!
  cout << s << endl;
  return 0;
}
```

Por ejemplo, si el usuario escribe `hello world` y pulsa Enter, `cin >> s` lee `"hello"`, y el `getline` que sigue lee `" world"` (nótese el espacio inicial) — el resto de esa misma línea — no la siguiente línea que el usuario pudiera escribir después. Tenga esto en cuenta siempre que mezcle `>>` y `getline` sobre el mismo flujo.

Observación interesante:

Un `string` puede cambiar su longitud, pero no mediante indexación: `s[i]` permite modificar un carácter que ya existe, nunca añadir ni eliminar uno. Para cambiar la longitud hay que usar las propias operaciones del string (`+=`, `.append()`, `.insert()`, `.erase()`...) o reasignarlo completamente. De todas ellas, `+=` es especialmente eficiente: un `string` mantiene algo de espacio de reserva (su *capacidad*) más allá de los caracteres que contiene en ese momento, de modo que la mayoría de las adiciones simplemente escriben en ese espacio a coste constante. Solo cuando se agota la capacidad el string asigna un buffer mayor y copia su contenido. Por tanto, para añadir caracteres de forma eficiente, use `+=` (o `.append()`), no `s = s + c`.

Medimos ambas formas de añadir un carácter `200 000` veces:

```
s += c     ->   0 ms
s = s + c -> 875 ms
```

Con `+=`, la capacidad crece a saltos (15⟶30⟶60⟶120⟶240… en la biblioteca del compilador GCC), aproximadamente duplicándose cada vez que se agota, no de carácter en carácter. Este *crecimiento amortizado* hace que `+=` sea $O(1)$ amortizado por adición. El factor de crecimiento exacto depende de la implementación.

Por el contrario, construir un string de n caracteres con `s = s + c` en un bucle es $O(n^2)$: cada iteración construye un string temporal completamente nuevo de tamaño $k+1$, copiando los k caracteres existentes, y luego lo mueve a `s`. No hay espacio de reserva que reutilizar. El trabajo total es:

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$$

Por eso `200 000` adiciones tardaron `875` ms en lugar de aproximadamente `0` ms. Duplicar n hace que el primer enfoque tarde aproximadamente el doble, y el segundo aproximadamente cuatro veces más.

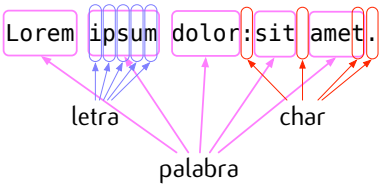
4·3.3 Ejemplos con cadenas de caracteres

La biblioteca de strings ofrece muchas más operaciones útiles. Varias de ellas son excelentes ejercicios de práctica, y así las usaremos. A continuación se presenta un recorrido breve por algunas de las más habituales, junto con lo que imprime cada línea:

```C++
// ... (assume #include <iostream>,
// #include <string>, using namespace std;, and this inside main())
string s = "Juan";
string s2(s), s3 = s2;                          // () is like =
cout << s << endl;                              // Juan
cout << s + " Falgueras" << endl;               // Juan Falgueras
        s += " (end)";
cout << s << endl;                              // Juan (end)
cout << "s.empty():     "    << s.empty() << endl;  // s.empty():    0
cout << "s.length():    "   << s.length() << endl;  // s.length():   10
cout << "s.max_size(): " << s.max_size() << endl;  // a huge,
                                                // implementation-defined
                                                // number
cout << s.append(" bye") << endl;               // Juan (end) bye
cout << s.find("end", 2) << endl;               // 6
cout << "At 6: " << s.at(6) << endl;            // At 6: e
cout << "At 6: " << s[6] << endl;               // At 6: e
cout << s.replace(6, 3, "hello") << endl;       // Juan (hello) bye
cout << s.insert(10, " and bye") << endl;       // Juan (hell and byeo) bye
cout << s.erase(10, 4) << endl;                 // Juan (hell byeo) bye
cout << s.substr(6, 9) << endl;                 // hell byeo
```

Observe que `.append()`, `.replace()`, `.insert()` y `.erase()` modifican `s` en su lugar y devuelven una referencia a `s` — por eso encadenarlos directamente dentro de `cout <<` funciona; `.find()` devuelve la posición de una coincidencia (o un valor especial de «no encontrado»); `.at()` y `operator[]` devuelven una referencia a un único carácter; y `.substr()` construye y devuelve un string completamente nuevo, dejando `s` intacto. Tómese el tiempo de rastrear a mano la posición y la longitud usadas en cada llamada, contrastándolas con el resultado impreso por la línea anterior — es una de las mejores formas de familiarizarse con la indexación base cero.

107: Cualquier carácter ASCII es un `char`; una letra es cualquier símbolo alfabético en mayúscula o minúscula `'a'-'z'`; una palabra es una secuencia de letras. Por tanto, los `char`s incluyen signos de puntuación como comas y puntos, así como caracteres de espacio en blanco: espacios, tabulaciones y saltos de línea



4·3.4 Punto de control

1) ¿Cuál es la salida del siguiente fragmento de código?

```
string name = "Juan";
name += name;
cout << name << endl;
```

2) ¿Cuál es el contenido de un string recién creado: `string s;` — y qué devuelve `s.length()` en ese momento?

3) Dado `string s = "programming";`, ¿qué valor devuelve `s.length()`, y qué posiciones de índice son válidas en `s[i]`?

4) ¿Es la siguiente una forma correcta de añadir un carácter a un string? Explique por qué sí o no, y muestre la forma correcta de hacerlo.

```
string s = "cat";
s[3] = 's';
```

5) Cuando el índice `i` está más allá del último carácter de `s`, ¿cómo se comporta `s.at(i)` de forma diferente a `s[i]`?

6) El usuario escribe `hello world` y pulsa Enter. ¿Qué contienen `first` y `rest` después, y por qué?

```
string first, rest;
cin >> first;
getline(cin, rest);
```

Tarea 11: Cadenas de caracteres

Los compiladores recientes no requieren la inclusión de la cabecera `<string>` porque la incluyen en `iostream`; por economía de sintaxis no la incluiremos, pero si tiene algún problema con su compilador y los strings no son reconocidos, simplemente use `#include <string>`.

En estos problemas utilizaremos extensamente la definición de **palabra**: cualquier secuencia contigua de letras *minúsculas* `['a'..'z']` (sin dígitos, sin espacios, sin signos de puntuación).

1 **Búsqueda de subcadenas.** Implemente un subprograma que imite el método `string::find` con la siguiente interfaz:

```
int find(string s, string pat);
```

Esta función debe devolver el índice de la primera aparición de `pat` dentro de `s`. Si el patrón no se encuentra, devuelva -1.

HINT: Construya un único bucle `while` que busque el j-ésimo carácter del patrón (es decir, `pat[j]`) incrementando `i`. Mantenga j en `0` salvo que `s[i+j]` sea igual a `pat[j]`, en cuyo caso incremente `j` manteniendo `i` fijo. Cuando `j` alcance la longitud del patrón, el patrón ha sido encontrado; en caso contrario, reinicie `j` e incremente `i`. › Sol

2 **Subcadenas indexadas.** Construya un subprograma que emule `string::substr` con la interfaz siguiente, pero usando índices que pueden ser negativos (contando hacia atrás desde el final):

```
string substrneg(string s, int from, int end=-1);
```

Si `from` o `end` son negativos, cuente desde el último carácter, donde -1 es el último. Si no se proporciona tercer argumento, toma por defecto el valor -1, es decir, todos los caracteres desde `from` hasta el final. Los caracteres en `from` y en `end` se incluyen ambos. › Sol

```
tryThis("abcdefgh", 2);       //  "cdefgh"                    C++
tryThis("abcdefgh", -2);      //  "gh"
tryThis("abcdefgh", -1);      //  "h"
tryThis("abcdefgh", -3, -2); //  "fg"
tryThis("abcdefgh", 4, 4);    //  "e"
tryThis("abcdefgh", 4, -2);   //  "efg"
tryThis("abcdefgh", 4, 3);    //  ""
tryThis("abcdefgh", 40, 50); //  ""
```

3 **Palíndromos.** Construya una función

```
string capitalit(string s);
```

que determine si la cadena es un palíndromo (se lee igual de izquierda a derecha que de derecha a izquierda). **Ignore** todo lo que esté fuera de `['a'..'z']` al recorrerla. Una buena colección de ejemplos se encuentra en palindromelist.net. › Sol

```
int main()                                                    C++
{
    void test(string s);

    test("");
    test("a");
    test("ab");
    test("aaab");
    test("A Santa dog lived as a devil God at NASA.");
    return 0;
```

```
}

void test(string s)
{
    bool isPalindromic(string s);

    if (isPalindromic(s))
        cout << s + ": IS palindromic" << endl;
    else
        cout << s + ": is NOT palindromic" << endl;
}
```

4 **A entero.** Construya una función:

```
int str2int(string s);
```

que analice la cadena `s` que contiene un número decimal y devuelva su valor como entero. Desde `main()`, pruebe con `"123"`, `"2021"`, `"01"` e imprima sus valores enteros: `cout << str2int("123") << endl;` › Sol

HINT: Comience con `result = 0` y recorra la cadena, sumando cada carácter como el valor entero que representa. Para el char `'3'` queremos el entero `3`, así que calcule `s[i] - '0'` (C++ permite restar chars directamente, usando sus códigos ASCII). Tenga en cuenta también que cada nuevo dígito desplaza el valor acumulado previamente en $10\times$: `res = 10 * res + s[i] - '0';`

5 Extienda `str2int` del ejercicio anterior para aceptar un segundo parámetro con la base (p. ej.
2 o 16). Para bases mayores que 10, los dígitos por encima del 9 se representan con letras A, B, C..., de modo que `'A' - 'A'` es 10: `res = base * res + tolower(s[i]) - 'a' + 10;` › Sol

Pruebe con:

```
int main()                                                    C++
{
    int str2int(string s, int base=10);

    cout << str2int("121231")       << endl;  // 121231
    cout << str2int("1111",   2)    << endl;  // 15
    cout << str2int("77",     8)    << endl;  // 63
    cout << str2int("A",     16)    << endl;  // 10
    cout << str2int("FFFF",  16)    << endl;  // 65535
    return 0;
}
```

6 Escriba una función `int firstInt(string s)` que reciba una cadena como `"You owe me 10, doesn't you?"` y devuelva el primer entero que contiene. Para ese ejemplo el resultado es `10`. › Sol

Pruebe con:

```
int main()                                                    C++
{
    int firstInt(string s);

    cout << firstInt("You owe me 10, doesn't you?") << endl; // 10
    cout << firstInt("No ten, but -12")             << endl; // 12
    return 0;
}
```

7 **Comparación.** Construya una función:

```
int strcmp(string s1, string s2);
```

que compare `s1` y `s2`, devolviendo

$$\begin{cases} -1 & \text{si } s_1 < s_2 \\ 0 & \text{si } s_1 = s_2 \\ +1 & \text{si } s_1 > s_2 \end{cases}$$

Tenga en cuenta que *avión* es menor que *tierra*, y *ex* es menor que *ejemplo*. › Sol Pruebe con:

```C++
int main()
{
    int strcmp(string a, string b);

    cout << strcmp("",          "")          << endl; //  0
    cout << strcmp("a",         "b")         << endl; // -1
    cout << strcmp("b",         "a")         << endl; //  1
    cout << strcmp("a",         "a")         << endl; //  0
    cout << strcmp("avion",     "avioneta") << endl; // -1
    cout << strcmp("ave",       "avioneta") << endl; // -1
    cout << strcmp("avioneta", "ave")       << endl; //  1

    return 0;
}
```

Necesitará comparar ambas cadenas carácter a carácter simultáneamente, comprobando que se mantiene dentro de los límites:

```
while ( i < int(a.length()) and i < int(b.length()) and a[i] == b[i] )
    ++i;
```

8 **Conteo de palabras.** Construya una función:

```
int noOfWords(string s);
```

que devuelva el número de palabras en una cadena. › Sol

Pruebe con:

```C++
int main()
{
    void tryWith(string s);

    tryWith(" I   Love   Programming");        // 3
    tryWith("  some; (like) a2b");             // 4
    tryWith(" live on time, - emit no evil "); // 6
    return 0;
}
```

9 **N-ésima palabra.** Implemente una función:

```
string nthWord(string s);
```

que devuelva la n-ésima palabra contenida en la cadena de entrada. › Sol

10 Construya una función

```
string capitalit(string s);
```

que devuelva la cadena de entrada con cada palabra en mayúscula inicial.
› Sol

```C++
int main()
{
    string capitalit(string s);

    cout << capitalit(" F.   of,   programming") << endl;
    // " F.   Of,   Programming"
    cout << capitalit("INDEPENDENTLY the case The Original Chars HaD")
    << endl;
    // "Independently The Case The Original Chars Had"
    return 0;
}
```

11 Escriba un procedimiento `coincidences()` que reciba dos parámetros de entrada: las cadenas `str1` y `str2` que contienen solo letras minúsculas. El procedimiento debe devolver dos valores de salida como parámetros: el número de letras en `str2` que coinciden con letras en las mismas posiciones en `str1`, y el número de letras en `str2` que aparecen en `str1` pero en posiciones diferentes.

Si un carácter aparece varias veces en una cadena pero solo una vez en la otra, se cuenta como 1. › Sol

Por ejemplo:

```C++
int main()
{
    void printOutput(string pat, string s);

    printOutput("ARXMBA", "RZXMKJH"); // -> 2, 1
    printOutput("JAKZEF", "AOA");     // -> 0, 1
    printOutput("JAKZAF", "AA");      // -> 1, 1
    printOutput("EA", "AA");          // -> 1, 0
    return 0;
}
```

12 **Scrabble.** Construya una función:

```
bool scrabble(string s1, string s2);
```

que determine si las dos cadenas son permutaciones una de la otra — es decir, tienen la misma longitud y cada carácter aparece el mismo número de veces. Por ejemplo, `"friendship"` y `"rifendihps"` son permutaciones una de la otra. Un enfoque útil para comprobar si dos cadenas son permutaciones una de la otra consiste en contar la frecuencia de cada letra. Escriba:

```
TFreq freqs(string s);
```

que devuelva la frecuencia de cada carácter ASCII en `s`, donde `typedef array<int,256> TFreq;`. Para comprobarlo, imprima cada carácter de `s` seguido de su frecuencia. Por ejemplo, para `"Charles' suit"`:

```Display
  1 ' 1 C 1 a 1 e 1 h 1 i 1 l 1 r 1 s 2 t 1 u 1
```

› Sol

13 Un **isograma** es una palabra sin letras repetidas, consecutivas o no consecutivas. Implemente una función que determine si una cadena que contiene solo letras es un isograma. Suponga que la cadena vacía es un isograma. Ignore las mayúsculas y minúsculas. › Sol

```
isIsogram("Dermatoglyphics") == true                    Ejemplos
isIsogram("aba")             == false
isIsogram("moOse")           == false  // ignore letter case
```

HINT: Use la solución del ejercicio 11 tras convertir las letras a minúsculas.

14 **Descomponer camelCase.** Construya una función que descomponga cadenas en camelCase insertando un guion bajo entre las palabras en minúsculas. Suponga que cada letra mayúscula inicia una nueva palabra.
› Sol

```
int main()                                                  C++
{
    string breakCamelCase(string s);

    cout << breakCamelCase("camelCasing") << endl; // camel_casing
    cout << breakCamelCase("__A__aA_A_")  << endl; // ___a__a_a__a_
    cout << breakCamelCase("camelC")      << endl; // camel_c
    cout << breakCamelCase("camElC")      << endl; // cam_el_c
    cout << breakCamelCase("APPLE")       << endl; // _a_p_p_l_e
    return 0;
}
```

15 Tiene una expresión aritmética restringida en una cadena como `"2+6-9-4"`, que contiene solo enteros, los operadores `+` y `-`, y espacios. La siguiente función la evalúa:

```
int express2int(string s)                                   C++
{
    int r = 0;
    for (int i = 0, sign = 1; i < int(s.size()); ++i) {
        if (s[i] == '+')
            sign = 1;
        else if (s[i] == '-')
            sign = -1;
        else if (s[i] >= '0' and s[i] <= '9')
            r += sign * (s[i] - '0');
    }
    return r;
}
```

Extienda el algoritmo para admitir números con más de un dígito y espacios.

HINT: Construya cada número como un `int` temporal hasta encontrar un nuevo signo; entonces use el temporal, reinícielo e interprete el nuevo signo. › Sol

Pruebe con:

```
int main()                                                  C++
{
    void test1(string s);
    void todo(string s);

    test1("2+6-4");          // 2+6-4: 4
    test1("6-4");            // 6-4: 2
    test1("0-1+1");          // 0-1+1: 0
    todo("2+634-4");         // 2+634-4: 632
    todo("  634   -  4 ");   //   634   -  4 : 630
    todo(" 123 ");           // 123 : 123
    todo(" -123 - 1");       // -123 - 1: -124
    return 0;
}
```

16 🧊 **Construir una cadena a partir de un entero.** Construya

```
string int2str(int n);
```

que convierta un entero en su representación como cadena.

HINT: Es fácil obtener el último dígito: calcule el resto módulo la base (`10` por defecto), luego sume `'0'` para obtener el char correspondiente. Divida entre `10` y repita hasta que el cociente sea `0`. El resultado está invertido, así que inviértalo antes de devolverlo. › Sol

17 Construya

```
string diff(string s, string t);
```

que reste los caracteres de `t` a los de `s`. Por ejemplo: `diff("telephone", "phonograph")` es `"telee"`, y `diff("holograph", "dog")` es `"hloraph"`.

HINT: Use la función `freqs` del ejercicio 11. › Sol

18 🧊 Existen dos tipos de comparación de patrones con comodines: globbing (usado en la comparación de archivos en Unix) y expresiones regulares. El globbing usa dos metacaracteres: `'*'` y `'?'`. Construya:

```
int glob(string s, string pat);
```

interpretando `'?'` en `pat` como *cualquier carácter individual*, de modo que `"exer01.cpp"` coincida con el patrón `"exer01.cpp"`.

HINT: Similar al ejercicio 7 (`strcmp`), pero en la comparación compruebe `pat[j] == '?' or s[i+j] == pat[j]`. › Sol

🧊🧊🧊 La solución completa (o casi completa) incluye `*` como posible parte del patrón, con el significado de *cero o más* caracteres en esa posición de la cadena. Por ejemplo en `glob("ejer01.cpp", "ejer*.cpp")` el `*` coincide con `01` › Sol

19 Una forma sencilla de extraer fragmentos de una cadena separados por un carácter dado es llamar a una función repetidamente hasta que devuelva una cadena vacía. Construya:

```
string nextPiece(string s, char sep='\0');
```

El segundo parámetro es el separador. En la primera llamada, proporcione el separador para inicializar un `static char theSep` y reiniciar el índice estático a `0`. Las llamadas sucesivas sin separador reanudan desde donde se dejó la llamada anterior. Las variables estáticas en funciones conservan sus valores entre llamadas. › Sol

Ejemplos

```
nextPiece("*abc *cde**fg*", '*') // -> ""
nextPiece("*abc *cde**fg*")      // -> "abc "
nextPiece("*abc *cde**fg*")      // -> "cde"
nextPiece("*abc *cde**fg*")      // -> ""
nextPiece("*abc *cde**fg*")      // -> "fg"
```

20 🧊 **Llaves válidas.** Escriba una función

```
bool validBraces(string s);
```

que reciba una cadena con llaves y determine si su orden es válido. Las cadenas de entrada pueden contener `(`, `)`, `[`, `]`, `{`, `}`. Una cadena es válida si cada llave de cierre cierra su correspondiente llave de apertura sin ninguna apertura sin cerrar en medio. Por ejemplo, `"is (a ) {valid}[strINg]"` y `"([{}])"` son válidas, mientras que `"not(}valid"` y `"[({})](]"` no lo son.

Ejemplos
```
validBraces( "(){}[]" ) => true
validBraces( "(}" )     => false
validBraces( "[(])" )   => false
validBraces( "([{}])" ) => true
```

HINT: Use una cadena como pila. Cada vez que encuentre una llave de apertura de `"([{"`, apílela al final de la cadena. Cada vez que encuentre una llave de cierre, debe corresponder a la última llave de apertura apilada. › Sol

21 **Validación EAN.** Un Número de Artículo Internacional (EAN) es un código de barras de 13 dígitos: 12 dígitos de datos seguidos de un único dígito de control. El dígito de control se calcula a partir de la cadena de 12 dígitos de la siguiente manera: se multiplica cada dígito en posición impar (1.ª, 3.ª, ...) por 1 y cada dígito en posición par (2.ª, 4.ª, ...) por 3, se suman los resultados y entonces: `checksum = 10 - (sum % 10)` (o `0` si `sum % 10 == 0`).

Por ejemplo, para `"400330101839"`:

Display
```
4x1+0x3+0x1+3x3+3x1+0x3+1x1+0x3+1x1+8x3+3x1+9x3
= 4+0+0+9+3+0+1+0+1+24+3+27 = 72
10 - (72 % 10) = 8  =>  EAN: 4003301018398
```

Construya una función que valide un código EAN dado, devolviendo `true` si es válido. Suponga que la entrada siempre tiene exactamente 13 dígitos. › Sol

Display
```
isValidEAN("4003301018398") => true
isValidEAN("4003301018392") => false
```

22 **Conversión RGB a hexadecimal.** La función

```
string rgb2hex(TRGB a);
```

recibe valores decimales RGB y devuelve la cadena hexadecimal correspondiente. Los valores decimales válidos son $0–255$; cualquier argumento fuera de ese rango debe recortarse al valor válido más cercano.

HINT: Escriba una función auxiliar que convierta un entero en su cadena hexadecimal de dos dígitos, rellenando con un cero inicial si es necesario. › Sol

Ejemplos
```
rgb2hex(TRGB{255, 255, 255}) // "FFFFFF"
rgb2hex(TRGB{255, 255, 300}) // "FFFFFF"
rgb2hex(TRGB{  0,   0,   0}) // "000000"
rgb2hex(TRGB{148,   0, 211}) // "9400D3"
```

23 La **distancia de Hamming** de dos cadenas de igual longitud es el número de posiciones en las que difieren — el número de sustituciones de caracteres necesarias para transformar una en la otra. Escriba:

```
int hamming(string a, string b);
```

Las cadenas contienen solo `0`s y `1`s. No es necesario validar la entrada. › Sol

Ejemplos

```
hamming("100101", "101001") == 2
hamming(  "1010",   "0101") == 4
```

24 🧊 **Patrones.** Escriba

```
bool wordPattern(string pattern, string str);
```

que compruebe si `str` sigue la misma secuencia que `pattern`. Cada letra del patrón representa una palabra distinta. › Sol

Ejemplos

```
wordPattern("abab", "truck car truck car");       // true
wordPattern("aaaa", "dog dog dog dog");           // true
wordPattern("abab", "apple banana banana apple"); // false
wordPattern("abab", "apple banana banana apple"); // true
wordPattern("aaaa", "cat cat dog cat");           // false
```

25 🧊🧊 **Mira y di.** La secuencia mira-y-di (explicada por Conway aquí) comienza:

ejemplo

```
1
11
21
1211
111221
312211
13112221
1113213211
```

Cada elemento se construye *leyendo* el anterior: cuenta cada secuencia de dígitos idénticos y di «cantidad dígito».

Construya

```
string lookandsay(string seed);
```

Partiendo de `seed = "1"`, itere e imprima los primeros 20 elementos.

Primero construya una función auxiliar `int countReps(string s, int pos)` que, empezando en `pos`, cuente cuántas veces `s[pos]` se repite consecutivamente. Convierta ese contador en un char (siempre `'1'`, `'2'` o `'3'`) y concaténelo con el dígito mismo para formar la siguiente secuencia. › Sol

26 🧊 **Números romanos.** Construya una función:

```
int rom2int(string s);
```

que analice la cadena `s` que contiene un número romano y devuelva su valor entero. Acepte tanto letras mayúsculas como minúsculas.

```
rom2int("iIi") // 3                                                    Display
rom2int("iv")  // 4
rom2int("liv") // 54
```

HINT: Leyendo la cadena al revés, solo hay que restar el valor de un dígito cuando es menor que el anterior. › Sol

27 Construya una función:

```
TNat sumNat(TNat a, TNat b);
```

donde `typedef string TNat;` sirve como contenedor de dígitos para números naturales arbitrariamente grandes.

```
int main()                                                             C++
{
    test("1", "1");
    test("122345", "98765");
    test("11112222333334444555566666777788889999",
         "11112222333334444555566666777788889999");
    return 0;
}
```

HINT: Invierta ambos números antes de sumarlos, use una variable `carry`, y luego invierta el resultado antes de devolverlo.

Usando las ideas desarrolladas hasta ahora, construya:

```
TNat multBy(TNat a, TNat b);
```

Compruebe su solución con:

```
25389450228724393822272644606314242417158103448 43                    Display
* 98464993182266617687544411784683307548280902514 =
24999720436728450547751653056583602336163680792674621614491391110567144360637821957437760563530 2
```

¿Puede usar sus funciones anteriores para construir una solución concisa para:

```
TNat pow(TNat b, TNat exp);
```

› Sol

Compruebe con:

```
25389450 ^ 9 =                                                         Display
438409178051912501457984911781933184948537741593458789062500000000 0
```

o O o

4·4 Estructuras o registros

Agrupar variables de distintos tipos bajo un único nombre permite a un programa manejar información compleja de forma jerárquica y posibilita modelar datos del mundo real de manera eficaz

108: Un `struct` es un contenedor de datos arbitrarios, con campos accedidos por sus `.nombres` y sin ningún orden implícito



Un `struct` o registro es un agregado de tipos de datos posiblemente distintos agrupados bajo un único nombre. Complementan lo que ofrece el array, en el sentido de que pueden contener diferentes tipos en la misma colección; pero también tienen una limitación que los arrays no tienen: su escalabilidad y la forma de acceder a ellos no son como las del array — se declara cada campo uno a uno, en lugar de dar un único contador total como con los arrays, y el acceso a los elementos, denominados *campos*, se realiza a través de sus propios nombres, por lo que los cómodos bucles `for` con índice ya no son posibles.

Para un número reducido de componentes no existe una gran diferencia entre ambos, ya que las «ventajas» y las «desventajas» desaparecen y la única diferencia que queda es la forma de acceder a las celdas/campos. Con las estructuras no hay un orden obvio para los campos en su uso, y cada campo se llama por su propio nombre.

Por otro lado, la sintaxis usada para acceder a los elementos va mucho más allá de los structs; puede encontrarse en lugares tan remotos como los localizadores de recursos de Internet que — aunque en orden inverso — expresan `detalleDe.Protocolo.máquina.lugar`, como en www.uma.es donde se accede al dominio español, dentro de él a la Universidad de Málaga y, dentro de ésta, al servicio web (`www`). Usaremos el mismo estilo sintáctico con los registros: del contenedor al contenido, usando puntos para indicar «pertenece a».

4·4.1 Definición. El tipo de datos struct

Para definir un `struct` hay que decidir primero los nombres y tipos de sus campos. La novedad (y quizás una inconsistencia en C++) es que, en lugar de `typedef`, se usa una nueva palabra reservada para crear el tipo: `struct`:

```
struct TName {
    type1 field1;
    type2 field2;
    type3 field3;
    …
};
```

Es posible (en un estilo anticuado) poner el nombre del nuevo tipo después de los campos, junto a la llave de cierre — nótese que con los registros hemos invertido el orden habitual, dando primero el nombre del nuevo tipo y después su composición. Hay una buena razón para esto: como bien saben los programadores experimentados, una larga lista de detalles dificulta encontrar la parte importante, en este caso el nombre del tipo. Por otro lado — no para el uso que haremos aquí, sino para un mecanismo importante que C++ ofrece para encadenar bloques en estructuras de longitud flexible como las listas enlazadas — colocar el nombre del struct antes de sus campos es mucho más conveniente,

ya que los campos pueden entonces referirse a él de forma recursiva a través de punteros.

Un ejemplo:

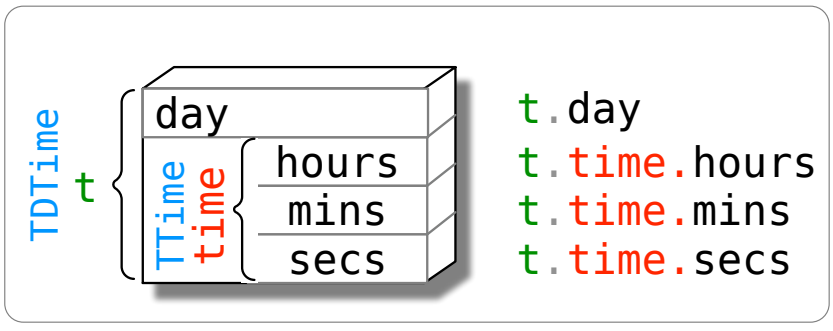
```
1  #include <iostream>
2  using namespace std;
3
4  // types
5  typedef string TMonth;
6
7  struct TDate {
8      int day;
9      TMonth month;
10     int year;
11 };
12
13 const TDate TODAY = {9,"Sept", 2026};
14
15 int main()
16 {
17     TDate birthD, currD;
18
19     birthD = TDate{1, "Jan", 2026};
20
21     cout << birthD.day << endl; // -> 1
22
23     return 0;
24 }
```

La inicialización de structs es sencilla: el único requisito es el orden de los campos (línea 13).

Para construir un valor de un tipo struct específico sobre la marcha, se antepone el nombre del struct a la lista de valores entre llaves — una conversión de estilo funcional (línea 19).

Acceder a los campos es fácil: solo hay que usar el punto, la clave genérica: `.` (línea 21)

Los structs pueden pasarse directamente como parámetros por valor o por referencia y devolverse de la misma manera. Se comportan exactamente como cabe esperar de variables básicas. En el siguiente ejemplo:



Las definiciones son:

```
struct TTime {
    int hours;
    int mins;
    int secs;
};
struct TDTime {
    int day;
    TTime time;
};
```

La notación de punto `.` a continuación del nombre de la variable proporciona acceso a sus miembros. Es como una llave para entrar en la caja de la estructura que le precede:

```
TDTime t;
t.day = 13;
t.time.hours = 2;
t.time.mins = 0;
t.time.secs = 20;
```

El contenido de esta variable sería: `{13,{2,0,20}}`. También podríamos inicializarla rápidamente con:

```
TDTime t = {13,{2,0,20}};
```

Alternativamente, si no se inicializa en el momento de la creación pero se necesita asignar un valor más tarde, se puede construir un valor literal convirtiendo el contenido:

```
TDTime t;
// ...
t = TDTime{13,{2,0,20}};
```

También es posible asignar valores iniciales por defecto a los campos de un struct — algo que los structs de C puro no podían hacer — lo cual resulta muy útil:

```
struct TList {
    TVec v;
    int n = 0;  // all TList new vars will
                //  start with this value here
};
```

Para acceder a sus elementos:

```
TList l;
l.v[0] = 33;
++l.n; // our list has now one value
```

De hecho, las clases de C++ no son más que extensiones de las definiciones de struct, a las que C++ añadió encapsulamiento, programación genérica y comportamiento a través de métodos — cuya sintaxis ya nos resulta familiar: `a.length()`, igual que cuando accedemos a un campo, con la diferencia de que estos son métodos, que pueden ser privados a la clase.

4·4.2 Operaciones sobre registros

Como tipos de datos arbitrarios, la única operación que se puede realizar sobre los `struct`s es la comparación, y solo para igualdad o desigualdad.

Como ejemplo de uso:

```C++
struct TTime {
    int hours, mins, secs;
};


int main()
{
    int toSecs(TTime t);

    TTime atime = {23, 30, 12};
    cout << "Secs: " << toSecs(atime) << endl;
    return 0;
}

int toSecs(TTime t)
```

```
{
    return t.hours * 3600 +
           t.mins * 60 +
           t.secs;
}
```

También se podría haber declarado `array<int, 3> TTime;` y referirse a los elementos con `atime[0]`, etc. Pero el uso de campos con nombre hace el programa mucho más legible y autodocumentado.

Con los structs se puede copiar su contenido como un todo a variables del mismo tipo, pero no se pueden comparar, como sí ocurre con los arrays. La razón no es difícil de adivinar. No se puede comparar `TRec a, b;` ~~`a == b`~~ ni ~~`a != b`~~ ni ~~`a > b`~~, etc.

4·4.3 Ejemplos con registros

En situaciones normales, los datos complejos requerirán estructuras complejas para representarlos. Habitualmente encontrará estructuras anidadas:

```
struct TDate {
    int day;
    TMonth month;
    int year;
};
struct TEmployee {
    int code;
    float salary;
    TDate joiningDate;
};
int main()
{
  TEmployee empl = {101, 2000, {19, "Aug", 1960}};
  empl.joiningDate.year++;
  printMonth(empl.joiningDate.month);
  //
}
```

Observe la cadena de puntos (líneas 14, 15).

4·4.4 Listas o secuencias de elementos

Para nosotros, un Array Abierto, o Lista, será un contenedor de un número indefinido de elementos. Vimos que la limitación más importante de los arrays es que son estáticos — no pueden crecer ni decrecer — por lo que son malos contenedores para cantidades de información de longitud variable. Especialmente en los lenguajes de scripting (JavaScript, Python, Ruby, etc.), los arrays son inherentemente flexibles en longitud (así como en el tipo de sus componentes), por lo que esta construcción detallada no sería necesaria; pero una forma sencilla de sortear el tamaño fijo de los contenedores en C++ es usar un array junto con un índice de ocupación. A veces esto ni siquiera es necesario, ya que se puede rellenar el array con un valor imposible (centinela) y tratar las celdas que lo contienen como vacías — pero ese no es un mecanismo generalmente válido y tiene inconvenientes. El siguiente enfoque es universalmente válido:

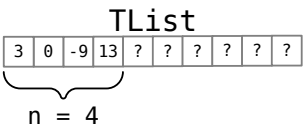
```
const int N = 100;
typedef array<int, N> TVector;

struct TOpenArr {
  TVector ar;
  int noElems = 0;
```

```
};
```

Tenemos un contenedor (inevitablemente con una capacidad máxima) pero una cantidad flexible de datos en su interior. Podemos insertar, eliminar o añadir elementos a esta lista; puede tratarse como un simple almacén, pero también como un array ordenado de elementos.

109: Una lista es un contenedor. Puede aprovechar sus propiedades de ordenación, pero también puede usarse simplemente como almacenamiento con capacidad variable




```
struct TList {
  int n = 0;
  TVec ar;
};
```

4·4.4.1 Punto de control

1) Defina un `struct TPoint` con `float x` y `float y`, y escriba el prototipo de una función `distance(TPoint a, TPoint b)` que devuelva la distancia entre dos puntos.
2) Dado `struct TBook { string title; string author; int year; float price; };`, escriba la declaración e inicialización de una constante `TBook` para un libro de su elección.
3) ¿Por qué dos variables `TBook` solo pueden compararse campo a campo «a mano», y no con `==`?
4) Diseñe un `struct TStudent` que anide un `TDate` (fecha de nacimiento) y un `TOpenArr` (de esta sección) que contenga las calificaciones de examen del estudiante.
5) ¿Qué ocurriría si intentara inicializar los campos de un `struct` dando un contador total, como se hace con `array<int, N>`?

Tarea 12: Estructuras

1 Responda a estas preguntas breves con respuestas cortas. Puede construir un archivo `.cpp` lleno de comentarios con las respuestas a cada una de ellas. X

1. Indique 2 diferencias entre estructuras y arrays,
2. ¿Cuál es la ventaja, si la hay, de declarar un `TComplex` como `struct` en lugar de como `array`? `TComplex` es una variable estructurada que contiene los 2 números reales que definen un número complejo matemático,
3. ¿Cuál es el inconveniente más importante de los arrays/matrices clásicos (es decir, `int a[3];`, `float m[5][4];`, o arrays/matrices no STL) que `struct` nunca ha sufrido?,
4. ¿Cuál cree que fue la inspiración sintáctica para la notación de puntos de Internet `https://www.uma.es/`? ¿El orden de los componentes es el mismo que el orden que usamos para acceder a los campos en nuestros `struct`s?,
5. ¿Por qué cree que, en general, las estructuras no pueden compararse entre sí?

2 Dadas las siguientes constantes, tipos y variables:

```C++
const int LLEN    = 50000;
const int SUBLLEN =    20;

typedef array<string,LLEN>    TList;
typedef array<string,SUBLLEN> TSublist;

enum TItemType {HARDW, SOFTW};
// Alternatively, we could declare:
//    typedef int TItemType;
//    const int HARDW = 0;
//    const int SOFTW = 1;

struct TRecList {
    TList list;
    TSublist sublist;
};

struct TCatalRec {
    string itemName;
    TItemType itemType;
    TRecList subcatal;
};

typedef array<TCatalRec,SUBLLEN> TCatal;

TCatal catal;
TCatalRec aCatal;
TRecList aList;
int count;
string aName;
```

¿Cuáles de las siguientes frases son correctas? (suponga que las variables ya tienen valores adecuados): X

1.
```C++
if (TCatalRec.itemType == HARDW)
    count++;
```
2. `catal[1].subcatal.list[2] = aCatal;`,
3. `catal[5].TRecList = aList;`,
4. `aCatal.itemName[2] = aList.list[2];`,

5. `catal[1].list.list[1][2] = aName[2];`,
6. `catal[19].list[1] = aName;`,
7.
```C++
if (catal[19].itemType == SOFTW) {
    catal[10].TRecList.sublist[3][9] = 'A';
}
```
,
8. `aCatal = catal[5];`,
9. `aCatal.subcatal = aList;`,

3 Queremos trabajar con números complejos

$$[\text{parte-real}] + i\,[\text{parte-imaginaria}]$$

por lo que definimos la estructura:

```C++
struct TComp {
    float re, im;
};
```

Escriba los tipos y subprogramas necesarios para el siguiente `main()`:

```C++
int main()
{
    TComp w = {2, 1};
    TComp z = {4, 3};

    printComp(sumComp(z, w));
    printComp(subComp(z, w));
    printComp(mulComp(z, w));
    printComp(divComp(z, w));

    return 0;
}
```

Si no recuerda las operaciones con números complejos, búsquelas en Wikipedia. › Sol

4 Defina las constantes y los tipos más apropiados para `TPatient` con:
- Un código identificador abierto (con números y letras).
- Un array de enteros con los `744` (24×31) valores de presión arterial correspondientes a cada hora de un mes.
- Un array con un máximo de `1 000` elementos reales con los niveles de expresión de un gen particular.
- El número total de expresiones génicas en el array anterior (siempre menor que `MAXGEN = 1000`).

A continuación, usando este tipo: › Sol

1. Construya una función que reciba un `TPatient` y devuelva la presión arterial máxima,
2. Construya una función que reciba un `TPatient` y devuelva, mediante dos parámetros por referencia, el día y la hora de la primera presión arterial máxima. Suponga que el día `0` hora `0` corresponde a la primera hora del primer día,
3. Construya una función que reciba un `TPatient` y un valor de presión arterial y devuelva el número de veces que el paciente tuvo una presión arterial igual o superior.

5 Defina las constantes y los tipos más apropiados para `TStudent` con:
- ID (cadena de caracteres).

- Un array de enteros con los identificadores de cada asignatura en la que el estudiante está matriculado.
- Un array de reales con las calificaciones correspondientes a cada una de las asignaturas anteriores.
- El número de asignaturas en las que está matriculado actualmente, siempre menor que `MAXNOSBJ = 50`.

A continuación:

1. Construya una función para obtener la calificación mínima entre todas las asignaturas matriculadas,
2. Construya una función para comprobar (verdadero/falso) si un `TStudent` supera un umbral `vmark` en todas las calificaciones,

Suponga que las asignaturas por debajo del número matriculado tienen todas una calificación válida, y que el estudiante se matricula en asignaturas de `0` al último de forma secuencial sin omisiones. › Sol

6 Un **array abierto** es una estructura que contiene un array (con un tamaño máximo estático `N`) y un entero que indica cuántos elementos están realmente almacenados (de `0` a `N`). › Sol

```C++
const int N = 100;
typedef array<int, N> TVector;
struct TOpenArr {
    int noelems;
    TVector ar;
};
int main()
{
    void printOArr(TOpenArr);
    TOpenArr sumOArr(TOpenArr, TOpenArr);

    TOpenArr a = {3, {{1, 2, 3}}};
    TOpenArr b = {3, {{3, 2, 1}}};
    printOArr(sumOArr(a, b));
    return 0;
}
TOpenArr sumOArr(TOpenArr a, TOpenArr b)
{ ... }
void printOArr(TOpenArr a)
{
    for (int i = 0; i < a.noelems; ++i)
        cout << a.ar[i] << " ";
    cout << endl;
}
```

Complete la función `sumOArr` que suma 2 arrays abiertos.

7 Existen dos formas de representar polinomios de una variable $p(x) = a_0 + a_1 x + a_{n-1} x^{n-1} + \ldots + a_n x^n$:

1. Un array simple de flotantes `array<float, N> TPoly` donde cada valor `a[i]` es el coeficiente a_i en $a_i x^i$. El índice corresponde al grado del monomio: `a[0]` es el término constante y `a[N-1]` es el coeficiente de x^{N-1}, así que `TPoly` puede contener polinomios de grado hasta $N-1$. Por ejemplo, el polinomio $x^5 - 3.25x^3 + 2.25x$ se representaría con `N = 6` como: `{{0, 2.25, 0, -3.25, 0, 1}}`. Todos los elementos del array contienen valores significativos. Algunos o todos pueden ser cero,
2. Un **array abierto** de pares de números —

`struct TMono {float coef; int deg;}` — cada uno almacenando el grado y el coeficiente de un monomio, ordenados de forma que el primer par tiene el mayor grado. El ejemplo anterior se convierte en◆:

◆En este caso un array abierto no es estrictamente *necesario* — un coeficiente cero indica que no hay más monomios. Se usa aquí para practicar con arrays abiertos.

grado	5	3	1	—
coef	1	−3.25	2.25	0

Para cada representación construya la estructura de datos y los siguientes subprogramas:

1. Imprimir un `TPoly`,
2. Añadir un `TMono` a un `TPoly`, modificando el último por referencia:
   `void addMono(TPoly& p, TMono m);`
   o alternativamente:
   `void addMono(TPoly& p, int degree, float coef);`
3. Añadir un `TPoly` a un `TPoly`, modificando el último por referencia:
   `void addPoly(TPoly& a, TPoly b);`

8 Usando el tipo `TPoly` del ejercicio anterior, construya una función que reciba un polinomio y un punto x y devuelva el valor del polinomio en ese punto:

```
float evalPoly(TPoly p, float x);
```

9 El **método de Horner** es una forma eficiente de evaluar polinomios en un punto x, evitando cálculos repetidos de potencias. En Wikipedia encontrará una solución en C. Proporcione dos soluciones mejores, una para cada una de las representaciones desarrolladas en el ejercicio anterior.

10 Construya una función que devuelva la **derivada** de un `TPoly`:

```
TPoly derivative(TPoly p);
```

Como puede leer en Wikipedia, la derivada de

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

es

$$n \cdot a_n x^{n-1} + (n-1) \cdot a_{n-1} x^{n-2} + \ldots + 2 \cdot a_2 x + a_1$$

11 Construya una función que devuelva la **suma** de dos valores `TPoly`:

```
TPoly sumPoly(TPoly a, TPoly b);
```

Sumar polinomios consiste simplemente en sumar sus coeficientes correspondientes. → Sol

12 Multiplicar un `TPoly` por un número real significa multiplicar cada coeficiente por el parámetro: › Sol

```
TPoly multBy(TPoly p, float b);
```

Multiplicar un `TPoly` por un monomio significa multiplicar cada coeficiente por `c` y añadir el grado del monomio `deg` a cada grado:

```C++
TPoly multBy(TPoly p, int deg, float c); // or
TPoly multBy(TPoly p, TMono m);
```

Una vez que tenga esas funciones, multiplicar un `TPoly` por otro `TPoly` consiste en sumar los polinomios obtenidos al multiplicar cada monomio de uno de ellos con el otro polinomio:

```
TPoly multBy(TPoly p, TPoly q);
```

La representación más sencilla es la forma **(a)** del ejercicio anterior: el array directo de coeficientes. Tras resolver las funciones auxiliares, la forma **(b)** basada en structs es más flexible.

13 Los compuestos químicos están formados por muchas moléculas idénticas. Una molécula es una secuencia (un array) de elementos, cada uno de los cuales es un átomo (p.ej. `Fe`, `H`, `Ni`) repetido un cierto número de veces. Cada elemento (`TElement`) almacena una cadena con el símbolo del átomo junto con su frecuencia (un entero en el rango 1–9). Una molécula (`TMolecule`) es un array abierto de valores `TElement`.

Escriba una función

```
TMolecule string2Molec(string s);
```

que analice la cadena de entrada y devuelva una `TMolecule`.

Escriba también un procedimiento para imprimirlas:

```
void printMol(TMolecule);
```

de la forma clásica, de modo que
`printMol(TMolecule{{"Na", 1}, {"O", 1}, {"H", 1}})`
imprima `NaOH`.

Pruébelo con estos ejemplos: › Sol

```C++
string2Molec("NaOH")
    // {{"Na", 1}, {"O", 1}, {"H", 1}}
string2Molec("Rb2SeO4")
    // {{"Rb", 2}, {"Se", 1}, {"O", 4}}
```

El símbolo de un átomo siempre comienza con una letra mayúscula; las letras adicionales son minúsculas. Para simplificar, suponga que la frecuencia es 1 cuando no sigue ningún dígito, y que en caso contrario está siempre entre 2 y 9.

o O o

4·5 Matrices

En Matemáticas y Física conocemos las matrices, frecuentemente bidimensionales, con las que se pueden expresar relaciones geométricas, interrelaciones, etc. También existen matrices tridimensionales e incluso de dimensiones superiores. En Geografía, Electrónica, Teoría de Juegos y en innumerables otras áreas, organizar los datos de esta forma es a menudo la única manera de modelar la información.

$$\begin{pmatrix} 10 & 2 & -3 & -7 & -2 & 4 & 5 & 5 & -3 & -1 \\ 4 & 10 & -2 & 6 & -1 & -7 & 6 & 2 & 9 & -8 \\ -5 & 1 & 3 & 9 & 3 & -2 & -2 & 8 & 9 & 4 \\ -7 & 2 & -3 & 7 & -1 & 4 & -4 & 8 & 9 & -3 \\ 9 & -8 & 3 & 0 & -9 & -7 & 1 & 4 & -4 & 8 \\ 9 & -8 & 9 & 1 & 2 & -9 & 8 & -8 & -4 & -4 \\ 3 & -6 & 4 & -2 & -1 & 1 & -6 & -6 & -8 & -3 \\ -3 & 8 & -5 & -6 & 9 & -7 & 2 & -7 & -2 & 1 \\ -3 & -9 & -4 & -5 & 0 & 5 & 5 & 0 & -9 & -7 \\ 2 & 8 & 10 & 9 & 0 & -1 & 9 & -1 & 5 & -4 \end{pmatrix}$$

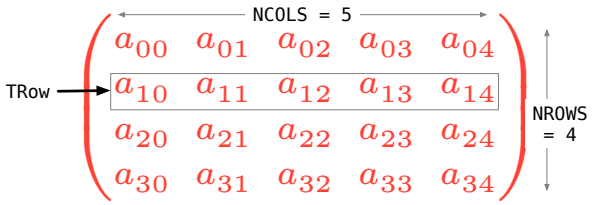
En C++ podemos construir matrices fácilmente — con una sintaxis que quizás resulta más sencilla que en otros lenguajes — como arrays de arrays:

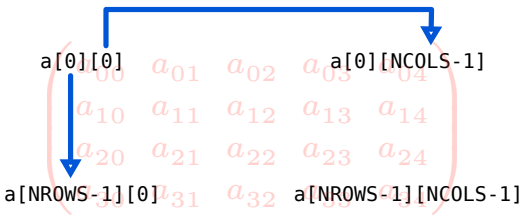
```C++
1 const int NROWS = 4;
2 const int NCOLS = 5;
3 typedef array< float,NCOLS> TRow;
4 typedef array<TRow,NROWS> TMatrix;
```

Así, en la línea 3 tenemos un array normal de floats que forma una única fila horizontal de la matriz, y en la línea 4 repetimos esas filas `NROWS` veces.

Cuando el tipo intermedio `TRow` no tiene utilidad en nuestro programa, podemos omitir su declaración y componer directamente la matriz de una sola vez:

```
typedef array< array<float, 5>, 4> TMatrix;
```





Cuando accedemos a los elementos de una matriz, debemos tener cuidado con el orden de los índices: el primero siempre se refiere a la fila (comenzando en `0` para la primera fila) y el segundo, a la columna:

```
const int NROWS = 4;
const int NCOLS = 5;
typedef array<array<float,NCOLS>,NROWS> TMatrix;

int main()
{
    TMatrix a;
    a[0][1] = 32.3;
```

Algunos lenguajes, aunque no el nuestro, permiten agrupar todos los índices de un array multidimensional en uno solo, ~~`[0,1]`~~.

Recuerda que también es posible acceder a una sola fila: `a[2]` te da la tercera fila. Independientemente de si la declaraste como su propio tipo, `typedef array< float,NCOLS> TRow;`, puedes pasar esa fila como parámetro o asignarla en bloque a otra fila — disponer de un tipo `TRow` con nombre simplemente facilita la escritura de esas declaraciones; la fila en sí se comporta exactamente igual de cualquier manera.

Recorrer una matriz es sencillo cuando el recorrido es completo, usando `for`s. Por ejemplo, para inicializar una matriz a `0`:

```
TMatrix m;

for (int r = 0; r < NROWS; ++r)
    for (int c = 0; c < NCOLS; ++c)
        m[r][c] = 0;
```

Matriz identidad:

```
const int N = 3;
typedef array<array<float,N>, N> TSqMat;

TSqMat identityMatrix()
{
  TSqMat m;
  for (int r = 0; r < N; ++r)
    for (int c = 0; c < N; ++c)
      if (r == c)
        m[r][c] = 1;
      else
        m[r][c] = 0;
  return m;
}
```

Para los ejercicios, la primera herramienta que necesitarás es

```
void printMat(TMatrix m)                                    C++
{
  for (int r = 0; r < NROWS; ++r) {
      for (int c = 0; c < NCOLS; ++c)
          cout << m[r][c] << " ";
      cout << endl;
  }
}
```

Inicializar mediante `cin >> m[r][c]` puede resultar tedioso, incluso para matrices pequeñas, especialmente cuando depuramos el código ejecutándolo repetidamente. Por ello recomendamos encarecidamente utilizar el método de inicialización rápida que se muestra a continuación:

```
const int NROWS = 4;                                        C++
const int NCOLS = 5;
typedef
 array<array<float,NCOLS>,NROWS> TMatrix;

int main()
{
    TMatrix a = {{
        { 1,  2,  3,  4,  5},
        { 6,  7,  8,  9, 10},
        {11, 12, 13, 14, 15},
        {16, 17, 18, 19, 20},
    }};
    //
}
```

Observa que aunque los arrays de números no requieren llaves dobles, el array de filas, al ser compuesto en lugar de números simples, requiere llaves dobles. Siempre puedes usar llaves dobles en ambos casos:

```
    TMatrix a = {{                                          C++
        {{ 1,  2,  3,  4,  5}},
        {{ 6,  7,  8,  9, 10}},
        {{11, 12, 13, 14, 15}},
        {{16, 17, 18, 19, 20}},
    }};
```

Recorrer una matriz es menos inmediato cuando, por alguna razón, necesitamos detenernos a mitad del recorrido:

```C++
bool isIn(TMatrix m, int x)
{
  bool found = false;
  int r = 0;
  while (r < NROWS and not found) {
    int c = 0;
    while (c < NCOLS and not found) {
      found = m[r][c] == x;
      ++c;
    }
    ++r;
  }
  return found;
}
```

No hagas esto, aunque pueda resultar tentador:

```C++
bool isIn(TMatrix m, int x)
{
  for (int r = 0; r < NROWS; ++r)
    for (int c = 0; c < NCOLS; ++c)
      if (m[r][c] == x) return true;
  return false;
}
```

Arrays multidimensionales: Siempre puedes extender las matrices a más dimensiones anidando un `array` más dentro del `typedef`:

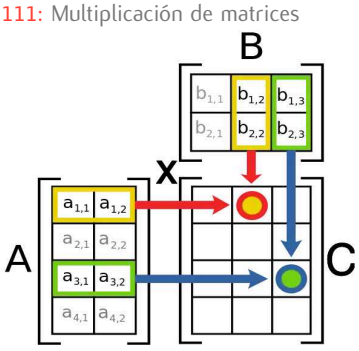
```
const int NROWS = 4;
const int NCOLS = 5;
const int NDEPTH = 3;
typedef array<array<array<float, NDEPTH>, NCOLS>, NROWS> TCube;

TCube cube;
cube[0][1][2] = 32.3;
```

Todo lo que hemos visto para las matrices — declaración, orden de acceso (índice más externo primero), recorrido con `for`s anidados — se extiende del mismo modo: un índice más, un bucle anidado más.

Dejamos el ejercicio de la multiplicación de matrices para el estudiante: $A \times B = C$, donde, como puede verse, una precondición es que el número de columnas de la primera matriz (A) debe coincidir con el número de filas de la segunda (B). Para simplificar, sin perder la generalidad del algoritmo, hazlo con matrices cuadradas (`TSqMat`):

111: Multiplicación de matrices




```
const int N = 3;
typedef array<array<float,N>, N> TSqMat;

TSqMat mult(TSqMat a, TSqMat b);
```

4·5.1 Punto de control

1) Escribe el prototipo de una función que devuelva la suma de todos los elementos de una `TMatrix`.
2) Escribe el prototipo de una función que devuelva la suma de una sola fila, dado su índice.
3) ¿Por qué `printMat` necesita que `NROWS` y `NCOLS` sean visibles, en lugar de recibirlos como parámetros?

4) ¿Qué significaría `cube[1][0][2]` para el tipo `TCube` definido anteriormente?

5) Necesitas un `bool isIn(TMatrix m, int x)` que se detenga en cuanto encuentre `x`. ¿Qué bucle debes usar — `for` o `while` — y por qué la versión tachada con `for` anterior no es aceptable?

6) Con `NROWS = 4` y `NCOLS = 5`, ¿a qué elemento hace referencia `a[3][4]`, y cuál es el rango válido de cada uno de los dos índices?

7) En la inicialización rápida `TMatrix a = {{ ... }};`, ¿por qué la lista exterior de filas necesita llaves dobles mientras que cada fila interior de números puede usar llaves simples?

8) Escribe el bucle o bucles necesarios para calcular la suma de los elementos de la diagonal principal (`m[i][i]`) de una `TSqMat m`. ¿Cuántos bucles son realmente necesarios?

Tarea 13: Matrices

$$M_{\text{NROW}=4 \times \text{NCOL}=5} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}$$

```
typedef array<array<int, NCOL>, NROW> TMat;

typedef array<array<int, N>, N>     TSqMat;
```

1 Construye un procedimiento

```
void printMat(TMat m)
```

que imprima la matriz `m` por pantalla. › Sol

2 Construye un procedimiento

```
void fillRandMat(TMat& m)
```

que rellene la matriz `m` con números aleatorios. Llama a `srand(time(0))` una sola vez para inicializar la secuencia aleatoria, y luego usa `rand()`, que devuelve un número pseudoaleatorio en $[0, \texttt{RAND_MAX}]$. Usa `printMat` del ejercicio anterior para mostrar el resultado. › Sol

3 Construye una función que devuelva una matriz cuadrada identidad: › Sol

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

```
TSqMat identSqMat();
```

4 Construye una función que determine si la `TSqMat` que recibe es **diagonal** (todos los elementos son `0` excepto los de la diagonal principal): › Sol

$$\begin{pmatrix} 33 & 0 & \cdots & 0 \\ 0 & -1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 33 \end{pmatrix}$$

```
bool isDiagonal(TSqMat m);
```

5 Construye una función que devuelva la **suma** de sus dos parámetros `TMat`: › Sol

```
TMat sumMats(TMat a, TMat b);
```

6 Construye una función que **transponga** la `TSqMat` que recibe: el elemento $[i, j]$ se intercambia con el elemento $[j, i]$.

```
TSqMat transpose(TSqMat a);
```

HINT: Ten cuidado de no intercambiar cada par de elementos dos veces. Recorre solo los elementos por encima de la diagonal principal (j desde $i+1$ hasta N). › Sol

7 Construye una función que determine si la `TSqMat` que recibe es **simétrica** (una matriz es simétrica si es igual a su propia transpuesta): › Sol

```
bool isSymmetric(TSqMat m);
```

8 Construye un procedimiento que intercambie dos filas de una `TMat`:

```
void swapRows(TMat& m, int r1, int r2);
```

Intenta declarar `TMat` como un array de `TRow` e intercambiar directamente los objetos fila correspondientes. › Sol

9 Construye un procedimiento que encuentre y devuelva las coordenadas (x, y) de la **primera celda máximo local** — aquella que es estrictamente mayor que todos sus vecinos existentes (8 vecinos en el interior, 5 en un

borde, 3 en una esquina). Recorre de arriba-izquierda a abajo-derecha y devuelve la primera que encuentre. Si no existe ninguna, devuelve -1 en ambas coordenadas.

```
void firstMax(TMat m, int& x, int& y);
```

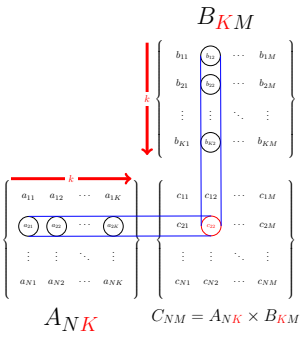
Usa el siguiente auxiliar, que devuelve `true` cuando el par de índices (i, j) está fuera de la matriz (de modo que los vecinos fuera de rango se tratan como menores que cualquier valor interior): › Sol

```C++
bool gt(TMat m, int v, int i, int j)
{
    return (i >= 0 and i < NROWS and j >= 0 and j < NCOLS) ?
                v > m[i][j] : true;
}
```

10 Multiplica dos matrices y devuelve el producto resultante:

```
TMatNM multMat(TMatNK a, TMatKM b);
```

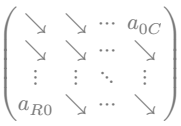
Recuerda que la celda (i, j) del resultado es el producto escalar de la fila i de la primera matriz por la columna j de la segunda. › Sol



Dado que aquí no usamos matrices cuadradas, se necesitan tres alias de tipo independientes:

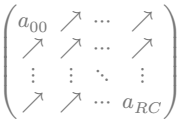
- `TMatNM` $(N \times M)$,
- `TMatNK` $(N \times K)$, y
- `TMatKM` $(K \times M)$.

11 Una **diagonal** de una matriz está formada por los elementos a los que se llega moviéndose diagonalmente: ambos índices aumentan juntos en las diagonales descendentes (↘), mientras que el índice de fila disminuye a medida que el índice de columna aumenta en las diagonales ascendentes (↗). Cada diagonal tiene una suma, y las diagonales varían en longitud.

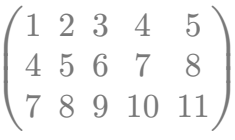


Encuentra la mayor suma diagonal de una matriz

```
typedef array<array<int, C>, R> TMat
```



Por ejemplo, dado:




```
int main()
{
    void test(TMat);
    test(TMat{{{{1, 2, 3,  4,  5}},
                {{4, 5, 6,  7,  8}},
                {{7, 8, 9, 10, 11}}}});
```

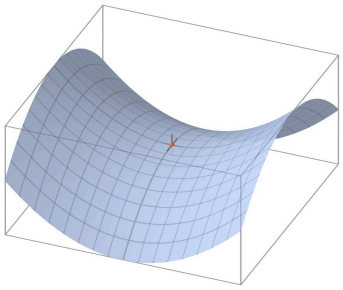
imprime la matriz y 21 como su `maxDiag`. Las sumas diagonales son: › Sol

```Display
Downward:  7,   4+8,  1+5+9,  2+6+10,   3+7+11,  4+8,    5
           7,    12,     15,      18,       21,   12,    5

Upward:    1,   2+4,  3+5+7,   4+6+8,    5+7+9, 8+10,   11
           1,     6,     15,      18,       21,   18,   11

Maximum sum: 21
```

12 🧊 Una `TMat` tiene un **punto de silla** en una celda que es un máximo local respecto a sus vecinos de fila y un mínimo local respecto a sus vecinos de columna, o viceversa.

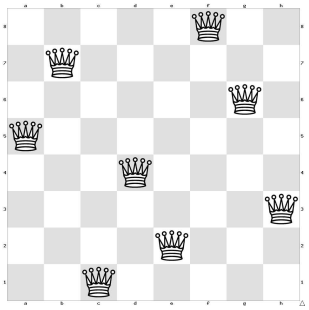


Un máximo local horizontal (o mínimo) es una celda estrictamente mayor (o menor) que sus vecinos izquierdo y derecho; en un borde, solo cuenta el único vecino existente. La misma definición se aplica verticalmente.

Una `TMat` puede tener más de un punto de silla. Construye una función que recorra toda la `TMat` y devuelva otra `TMat` del mismo tamaño rellena con:

- `+1` donde haya un punto de silla máximo-de-fila / mínimo-de-columna,
- `-1` donde haya un punto de silla máximo-de-columna / mínimo-de-fila,
- `0` en el resto › Sol

13 🧊 Un tablero de ajedrez es una `TSqMat` con $N = 8$. Cada jugador tiene 16 piezas: un rey, una reina, dos torres, dos caballos, dos alfiles y ocho peones — representados convenientemente con un `enum` de C++ o constantes enteras. Sin embargo, este ejercicio se centra en un problema diferente basado en el ajedrez: el problema de las ocho reinas. El objetivo es colocar ocho reinas en un tablero 8×8 de forma que ninguna reina amenace a otra. Construye una función que determine si una disposición dada está libre de amenazas:



```
bool noThreaten(TSqMat m);
```

En el tablero `TSqMat`, `true` significa que una reina ocupa esa celda. Para cada celda que contenga `true`, verifica que todas las celdas accesibles desde ella (misma fila, misma columna, ambas diagonales) estén libres de reinas. › Sol

14 🧊🧊 **Cuadrados mágicos**

Un cuadrado mágico es una `TSqMat` de $N \times N$ enteros **distintos** (cada uno usado exactamente una vez) en la que las sumas de cada fila, cada columna y ambas diagonales principales son iguales. Este valor común es la **constante mágica** $\frac{n(n^2+1)}{2}$. (Hay una fórmula más completa disponible aquí.)

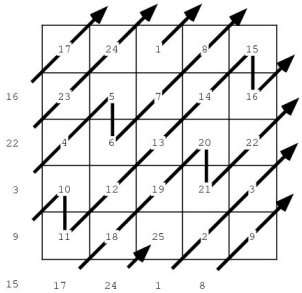
$$\begin{pmatrix} 2 & 7 & 6 \\ 9 & 5 & 1 \\ 4 & 3 & 8 \end{pmatrix}$$

Los cuadrados mágicos existen para todo N excepto $N = 2$. El ejemplo más antiguo conocido, el único cuadrado mágico 3×3 que data del 650 a.C., es:

$$\begin{pmatrix} 23 & 28 & 21 \\ 22 & 24 & 26 \\ 27 & 20 & 25 \end{pmatrix}$$

Sumar 19 a cada elemento produce otro cuadrado mágico válido:

El algoritmo de H. Coxeter construye cuadrados mágicos para cualquier N impar:



- Comienza colocando el 1 en la celda central de la fila superior.
- Muévete diagonalmente hacia arriba y a la derecha, colocando números consecutivos en celdas vacías.
- Al cruzar un borde, envuelve al lado opuesto (como si la matriz estuviera repetida infinitamente).
- Si la celda de destino ya está ocupada, muévete una celda hacia abajo en su lugar y continúa.

Implementa `void coxeter(TSqMat& magic);` Prueba con valores impares de N y muestra los resultados. › Sol

15 Recorrido del caballo de ajedrez

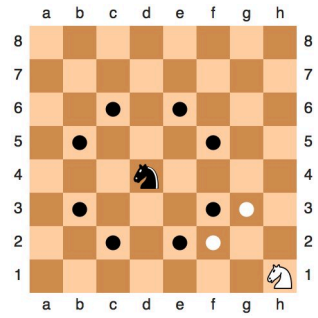
Un recorrido del caballo es una secuencia de movimientos de un caballo en un tablero de ajedrez tal que el caballo visita cada casilla exactamente una vez.

Este ejercicio no pide que **encuentres** dicho recorrido; en cambio, debes **verificar** si un recorrido dado es correcto. › Sol

El recorrido está codificado en una `TSqMat` de enteros (asume $N = 8$): la celda numerada con 1 es la casilla inicial, el caballo debe aterrizar después en la celda numerada con 2 mediante un salto legal, y así sucesivamente. Un recorrido completo y válido cubre las $8 \times 8 = 64$ casillas. Construye:

```
bool okTour(TSqMat m);
```

Un caballo puede saltar a un máximo de 8 posiciones: dos casillas horizontalmente y una verticalmente, o dos casillas verticalmente y una horizontalmente, en cualquier combinación de direcciones.



Un ejemplo de recorrido válido:

```
TSqMat m = {{                                                    C++
    {{ 1, 64, 61, 42, 37, 40, 55, 52}},
    {{60, 43,  2, 63, 56, 53, 36, 39}},
    {{ 3, 62, 59, 20, 41, 38, 51, 54}},
    {{44, 19,  4, 57, 50, 21, 32, 35}},
    {{ 5, 58, 45, 18, 31, 34, 49, 22}},
    {{14, 17,  6, 25, 46, 11, 30, 33}},
    {{ 7, 26, 15, 12,  9, 28, 23, 48}},
    {{16, 13,  8, 27, 24, 47, 10, 29}}
}};
```

$$\begin{pmatrix} 1 & 64 & 61 & 42 & 37 & 40 & 55 & 52 \\ 60 & 43 & 2 & 63 & 56 & 53 & 36 & 39 \\ 3 & 62 & 59 & 20 & 41 & 38 & 51 & 54 \\ 44 & 19 & 4 & 57 & 50 & 21 & 32 & 35 \\ 5 & 58 & 45 & 18 & 31 & 34 & 49 & 22 \\ 14 & 17 & 6 & 25 & 46 & 11 & 30 & 33 \\ 7 & 26 & 15 & 12 & 9 & 28 & 23 & 48 \\ 16 & 13 & 8 & 27 & 24 & 47 & 10 & 29 \end{pmatrix}$$

16 Proyecto — Juego de la vida

El juego de la vida de Conway es un autómata celular en el que una matriz de células evoluciona a través de pasos de tiempo discretos (generaciones) basándose únicamente en los estados de las células vecinas.

El universo es una cuadrícula bidimensional infinita; cada célula está `VIVA` o `MUERTA` e interactúa con sus 8 vecinos (adyacentes horizontal, vertical y diagonalmente). En cada generación se producen simultáneamente las siguientes transiciones:

1. Una célula viva con **menos de 2** vecinos vivos muere (subpoblación).
2. Una célula viva con **2 o 3** vecinos vivos sobrevive.
3. Una célula viva con **más de 3** vecinos vivos muere (sobrepoblación).
4. Una célula muerta con **exactamente 3** vecinos vivos nace (reproducción).

Resumido en una tabla:

vecinos	estado	→	nuevo estado
< 2	VIVA	→	MUERTA
$= 2, 3$	VIVA	→	VIVA

vecinos	estado	→	nuevo estado
$= 3$	MUERTA	→	VIVA
> 3	VIVA	→	MUERTA

En un programa sencillo de C++, la cuadrícula puede ser una `TMat` no mayor que una ventana de terminal típica (unas `40` filas × `100` columnas). Inicialízala con `rand()` como en el ejercicio 2, y redibuja cada generación tras limpiar la pantalla enviando:

```
const string CLRSCR = "\x1B[H\x1B[J";
```

(En Windows puede ser necesario usar `system("cls")` en su lugar.)

El subprograma principal es:

```
void nextGeneration(TMat& m)
```

Para evitar que los estados anteriores interfieran con la siguiente generación, **modifica el array en su lugar** usando valores de estado temporales para cada célula durante el paso de evaluación. Un segundo paso escribe entonces los valores definitivos. Los estados de las células pueden representarse con:

```
typedef int TState;
const int DEAD     = 0;
const int ALIVE    = 1;
const int TODIE    = 2;
const int TOBEBORN = 3;
```

Las células del borde pueden tratarse como rodeadas de células muertas, o puedes **envolver** los índices de modo que los bordes opuestos sean vecinos (como si la cuadrícula estuviera doblada en un toro). › Sol

Tras mostrar una generación, el programa puede esperar una pulsación de tecla o pausar automáticamente con `usleep(100000)` (`100` ms) antes de dibujar la siguiente. Un auxiliar para rellenar la cuadrícula aleatoriamente:

```
void arrAleat(TMat& a)
{
    for (int i = 0; i < NROWS; ++i)
        for (int j = 0; j < NCOLS; j++)
            a[i][j] = (TState)(rand() % 2);
}
```

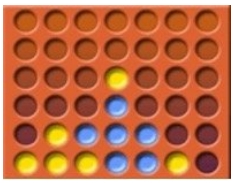
17 *(Del examen)* Diseña un programa, organizado en subprogramas, que lea del teclado un número natural `M` (que debe ser mayor que `0` y como máximo una constante conocida, p. ej. `MAX = 10`). El programa lee entonces una secuencia indefinida de enteros **distintos** que termina en `0` y calcula la media de los `M` números más grandes de la secuencia. Si la secuencia tiene menos de `M` números, calcula la media de los que se hayan introducido. › Sol

Display

```
M = 4
7 8 1 2 3 4 5 6 0  ->  (5 6 7 8)  6.5

M = 5
1 2 3 0  ->  (1 2 3)  2
```

18 *(Del examen)* **Cuatro en raya**



El juego se desarrolla en un tablero vertical con 6 filas y 7 columnas. Una ficha introducida en una columna cae a la celda vacía más baja de esa columna. Dos jugadores se turnan, cada uno con fichas de diferente color (representadas aquí como `1` y `2`; `0` significa celda vacía). El primer jugador que forme una línea de cuatro fichas del mismo color — horizontal, vertical o diagonalmente — gana.
Diseña la estructura de datos para el tablero e implementa los siguientes dos subprogramas. Para probarlos, escribe un `main()` que inicialice un tablero vacío, realice varias llamadas e imprima los resultados (incluido el contenido completo del tablero). › Sol

a. `insertToken` — apila una ficha en una columna:

```
void insertToken(TBoard& tab, int token, int col,
                 bool& ok, int& row);
```

Si la columna está llena, devuelve `false` en `ok`; en caso contrario devuelve `true` y la `row` en la que se colocó la ficha. El valor de la ficha y el número de columna son siempre válidos.

b. `fourInArow` — comprueba eficientemente si la última ficha añadida crea una línea ganadora:

```
bool fourInArow(TBoard tab, int row, int col);
```

Devuelve `true` si `tab` contiene cuatro fichas iguales en línea (horizontal, vertical o diagonal) una de las cuales está en la posición `(row, col)`. Devuelve `false` si no hay ninguna ficha en esa posición. Los valores de fila y columna son siempre válidos.

19 La **distancia entre dos apariciones de una palabra** en un texto es el número de palabras que aparecen entre ellas. Diseña un programa que lea palabras del teclado hasta que se introduzca la palabra `end`, y luego imprima la distancia máxima entre cada par de palabras repetidas. Las palabras que aparecen solo una vez no se imprimen. › Sol

Display

```
Enter text (end to finish):
por favor traeme el abrigo que tengo en el armario
que tengo que ir por el descampado a por el pan end

Maximum distances between repeated words:
  por   : 13
  el    :  6
  que   :  4
  tengo :  4
```

Notas:
- El texto contiene un número indefinido de palabras.
- Termina con la palabra centinela `end`.
- Cada palabra consiste únicamente en letras minúsculas y tiene longitud indefinida pero acotada.
- Las palabras están separadas por espacios.
- Aparecerán como máximo `MAX_DIFF_WORDS` (una constante conocida) palabras distintas en el texto.

o O o

Tarea 14: Problemas de exámenes finales

Tras revisar los contenidos del primer tema, responde a las siguientes preguntas. Puedes registrar tus respuestas en un archivo llamado `02.theoryQuestions.txt` enumerando el número de pregunta seguido de tu respuesta. Asegúrate de incluir 1) el nombre del archivo 2) tu nombre/alias y la fecha al principio del archivo. Recuerda subir el archivo al Dropbox de la tarea correspondiente en Campus Virtual.

1 Un programa lee un entero `pattern` positivo (volviendo a preguntar si ≤ 0), luego una secuencia de enteros que termina en `0`, e imprime todos los enteros positivos de la secuencia cuya suma de dígitos coincida con la de `pattern`. Los negativos se descartan; no se repiten en la salida. › Sol

- Construye `sumDigits()` para calcular la suma de dígitos de un entero positivo.
- Lee los enteros e imprime aquellos cuya suma de dígitos coincida con `pattern`.

A tener en cuenta:
- La modularización y la elección correcta del bucle son importantes; se evalúa la eficiencia.
- La secuencia es ilimitada, terminada en `0` (no incluido).
- Los enteros están separados por espacios o saltos de línea.
- El número máximo de enteros distintos que coincidan es `NMAX_DIFF` (p. ej. `50`).

```
Enter the pattern: -3                                          Display
Enter the pattern: 0
Enter the pattern: 153
Enter integers (0 to end):
1242 564 -6576 9 1242 1523 7689713 3 -531
Result: 1242 9 45
```

2 Desde `main()`, llama a `test()` que recibe una matriz de enteros $R \times C$ (`R=4`, `C=3`). Muestra la matriz con `display()`, luego llama a `firstCent()` para encontrar y devolver los índices de la primera celda igual a la suma de sus vecinos (3, 5 u 8 vecinos, según la posición). Devuelve $(-1, -1)$ si no se encuentra ninguna. Se evalúa la eficiencia. › Sol

$$M = \begin{pmatrix} 3 & 1 & 2 \\ 2 & 16 & 2 \\ 1 & 3 & 2 \\ 5 & 1 & 6 \end{pmatrix}$$

- $M_{0,0} : 3 \neq \sum\{1, 2, 16\} = 19$.
- $M_{1,1} : 16 = \sum\{3, 1, 2, 2, 2, 3, 1, 2\} \longrightarrow$ devuelve $(1, 1)$.

A tener en cuenta:
- Se evalúa la modularización. Puedes añadir subprogramas.
- Los valores pueden ser negativos o cero.
- Inicializa rápidamente y pasa las matrices como parámetros; no las leas del teclado.

Display
```
Matrix:
3 1 2
2 16 2
1 3 2
5 1 6
First cell satisfying the condition: (1, 1)

Matrix:
2 1 7
2 5 8
2 2 1
7 7 2
First cell satisfying the condition: (-1, -1)

Matrix:
2 1 7
2 5 8
2 2 1
7 6 9
First cell satisfying the condition: (3, 2)
```

3 Un perfil de montaña se almacena como un array de `N=10` alturas enteras. Por ejemplo:

Display
```
                    *
                *   *
            *   *   *
*           *   * * *
*           * * * * *
* *       * * * * * *
* * *     * * * * * *
* * * * * * * * * *
5 3 2 1 3 6 4 7 5 8
```

Tiene picos en los índices: 0 (altura 5), 5 (altura 6), 7 (altura 7) y 9 (altura 8). Una posición es un pico cuando ambos vecinos son más bajos; los extremos son picos cuando su único vecino es más bajo. Las distancias entre picos consecutivos son: $5-0=5$, $7-5=2$ y $9-7=2$.

Construye `peaksDists()` que reciba un array de alturas y devuelva un array terminado en cero con las distancias entre picos consecutivos. Para el ejemplo anterior: 3 distancias `{5, 2, 2, 0}`, impresas como: `Distance between a peak and the next: 5 2 2` › Sol

4 La **distancia entre dos palabras** en un texto es el número de palabras que hay entre ellas: entre `a` y `d` en `"a b c d"` la distancia es 3. Lee una secuencia de palabras que termina en `end` e imprime la distancia máxima entre palabras repetidas. Las palabras no repetidas no aparecen en la salida. flux › Sol

```
Type in the words (end to finish):                                    Display
can you can a can as a canner can can a can end

Max distance between repeated words:
a: 4
can: 4
```

Observaciones:
- Todas las palabras están en minúsculas; solo hay espacios entre ellas.
- La longitud de la secuencia es ilimitada.
- El total de palabras distintas está limitado a `MAXPAL` (constante conocida).

5 Lee un número natural `m` con `0 < m <= MAXDIFNUMS` (volviendo a preguntar hasta que sea válido), luego una secuencia de números naturales que termina en `0`. Muestra los `m` números más grandes junto con las **posiciones en las que aparecieron**. Si se introdujeron menos de `m` números, muéstralos todos. › Sol

```
Type in m: 5                                                          Display
Type in a sequence of natural numbers, 0 ends it:
2 3 4 7 28 4 5 1 1 1 9 7 4 4 28 2 1 3 3 6 0
The 5 greatest ones and their positions are:
   7:  4 12
  28:  5 15
   9: 11
   5:  7
   6: 20
```

Estructuras de datos y prototipos de partida:

```cpp
const int MAXDIFNUMS = 50;
const int MAXREPETIT = 10;

typedef array<int,MAXREPETIT> TPoss;
struct TNum {
  int num;
  TPoss pos;
};
typedef array<TNum,MAXDIFNUMS> TSeq;


int main()
{
  void addNumbers(TSeq& seq, int m);
  void displayNumbers(TSeq seq);
  void init(TSeq& seq);
  TSeq seq;
  init(seq);
  int m = 5;
  addNumbers(seq, m);
  displayNumbers(seq);
  return 0;
}
```

Subprogramas sugeridos:

```
int lenSeq(TSeq seq);
int posn(TSeq seq, int n);
int posMin(TSeq seq);
void initPos(TPoss& poss);
```

```
void writeFirstTime(TSeq& seq, int i, int n, int pos);
void addPos(TSeq& seq, int i, int pos);
```

6 Un **sopa de letras** es una matriz de caracteres $N \times N$. Construye un subprograma que reciba la matriz y una cadena, y devuelva si la cadena fue encontrada y, en caso afirmativo, su posición inicial y dirección (`NORTH`, `SOUTH`, `EAST`, `WEST`). La palabra aparece como máximo una vez. Construye también `main()` para leer la matriz (8 cadenas de 8 caracteres, `N=8`), leer la cadena de búsqueda, llamar al subprograma e imprimir el resultado o `"it is not there"`. › Sol

```
8 strings with 8 chars each:                                    Display
  wynapjgu
  fplmodrd
  ipiankee
  riurkxrd
  cjogomid
  ikgoxpiq
  lcdrghef
  sgnpycft
Enter a word to find: program
The word "program" is in the position (7,3) North
```

HINT: Para rellenar la matriz, pasa un parámetro de cadena con todos los caracteres de la matriz sin espacios ni saltos de línea, y luego asigna los caracteres a las celdas de la matriz secuencialmente. Después puedes pedir una palabra a buscar.

7 Define `TList` (que contiene hasta `MAX` enteros). Implementa `sieve()` que toma una `TList list1` y un entero `n` como parámetros, devolviendo una `TList list2` que contiene solo los números de `list1` que aparecen **exactamente** `n` veces, sin duplicados. › Sol

```
list1 is: 1 3 4 3 1 3 0 -6 4                                    Display
n is: 2
Then list2 will be: 1 4
```

8 Trabaja con **matrices incompletas** `TMat` de tamaño máximo `MAX×MAX`: › Sol

```
const int MAX = 10;
struct TMat {
    int nrows, ncols;
    array< array<int,MAX>, MAX> data;
};
```

Construye procedimientos para leer e imprimir `TMat`. Luego, construye un subprograma para **multiplicar** dos matrices y obtener una tercera. Pide al usuario las dimensiones antes de leer los elementos. Supón que las matrices son multiplicables (columnas de la primera = filas de la segunda); no compruebes esta precondición.

Para multiplicar dos matrices, debemos multiplicar cada fila de la primera matriz por cada columna de la segunda. Son multiplicaciones **escalares**, por lo que debemos multiplicar cada elemento correspondiente y **sumar**

estos productos. Es decir: dados $\boldsymbol{A} = a_{ij}$ y $\boldsymbol{B} = b_{ij}$, el producto $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$ tiene elementos:

$$C_{ij} = \sum_{k=0}^{M} A_{ik} \cdot B_{kj}$$

Para hallar el elemento C_{ij}, iteramos sobre un índice k que recorre la dimensión común M, igual tanto al número de columnas de la primera matriz como al número de filas de la segunda.

9 El **centro** c de un vector v de N valores naturales es el índice

$$1 \leq c \leq N-2$$

que satisface: › Sol

$$\sum_{i=0}^{c-1}(c-i) \cdot v_i = \sum_{j=c+1}^{N-1}(j-c) \cdot v_j$$

Construye `centreOfVector()` que reciba `v` y devuelva dos valores por referencia: si existe un centro y, en caso afirmativo, su índice.

Display
```
vec is: 6 2 3 0 1
Then centre is 1   // 1*6 = 6  IS EQUAL TO  1*3 + 2*0 + 3*1 = 6

vec is: 1 2 1 1 0
No centre
```

HINT: Puedes llamar a un procedimiento `test(TVec v)` dos veces desde `main()`, una con cada uno de estos vectores `v`, y mostrar los resultados ahí.

10 Lee una cadena `suffix` y una secuencia de palabras, algunas de las cuales terminan con ese sufijo. Imprime todas las palabras que terminan con dicho sufijo, sin repeticiones.

Display
```
Suffix?: ing
Type in a sequence of words (end to finish):
i decided going hiking instead of going riding bike end

Words ending in "ing":
going hiking riding
```

Observaciones: › Sol

- Todas las palabras están en minúsculas; solo hay espacios entre ellas.
- El número de palabras y la longitud de la secuencia son ilimitados.
- El total de palabras distintas está limitado a `MAXPAL` (constante conocida).

11 Diseña:

```
int noOcurrs(TSeqNum seq, TPerm aPermutation);
```

`TSeqNum` es un array de `SEQ_LEN=11` enteros; `TPerm` es un array de `PERM_LEN=4` enteros. `noOcurrs()` devuelve cuántas veces `aPermutation`

aparece como permutación de una porción del mismo tamaño de `seq`, deslizando `aPermutation` a lo largo de `seq`.

Para comprobar si una porción de `seq` es una permutación de `aPermutation`, construye:

```
int countFrom(TSeqNum a, int from, int x);
```

que devuelve cuántas veces aparece `x` en `a` a partir de `from` durante `PERM_LEN` posiciones. En el ejemplo, `noOcurrs()` devuelve 2. › Sol

12 La **codificación por longitud de ejecución** comprime cadenas con largas repeticiones de caracteres. `rle()` sustituye las ejecuciones de caracteres iguales por el carácter una vez, seguido de `/` y el recuento — pero solo cuando la ejecución supera `THRESHOLD`. Las ejecuciones más cortas se copian sin cambios.

Con `THRESHOLD=4`, se produce la siguiente compresión:

abcaaaaaaadbbbbbbdddvvvvxxaaaaaaaaccc
abca/7db/6ddd v/4xxa/8ccc

Construye:

```
string rle(string s);
```

Como primera simplificación, supón que las longitudes de ejecución son siempre menores que 10. (Para ejecuciones más largas se necesitaría un carácter delimitador adicional.) › Sol

13 Construye `sliding()` (completa su interfaz) que reciba dos cadenas y dos enteros por referencia. En el primer entero por referencia devuelve el desplazamiento más pequeño de la segunda cadena sobre la primera que produce el mayor número de coincidencias de caracteres, y a través del último entero por referencia devuelve ese número de coincidencias.

Para `acbaabch` y `aabghc`: desplazamiento 3, coincidencias 4. › Sol

Display

```
acbaabch  // displ: 0; coinc: 2
aabghc
acbaabch  // displ: 1; coinc: 1
 aabghc
acbaabch  // displ: 2; coinc: 1
  aabghc
acbaabch  // displ: 3; coinc: 4
   aabghc
...
acbaabch  // displ: 7; coinc: 0
       aabghc
```

14 Construye:

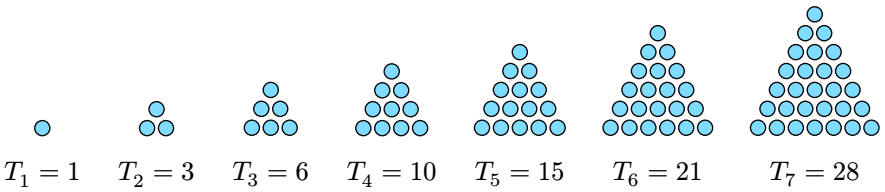
```
void clockwise(TSqMat& m);
```

donde `TSqMat` es una matriz cuadrada de tamaño `N`. `clockwise()` rota la matriz `m` 90° en sentido horario **en su lugar** sin usar variables auxiliares.
› Sol

HINT: descompón la operación en dos pasos, siendo el primero **transponer**.
Para `N=5`:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix} \xrightarrow{\texttt{clockwise()}} \begin{pmatrix} 21 & 16 & 11 & 6 & 1 \\ 22 & 17 & 12 & 7 & 2 \\ 23 & 18 & 13 & 8 & 3 \\ 24 & 19 & 14 & 9 & 4 \\ 25 & 20 & 15 & 10 & 5 \end{pmatrix}$$

15 Se dice que un número es el k-ésimo *número triangular* cuando es igual a la suma de los primeros k números naturales: $T_k = 1 + 2 + ... + k$.



En el **solitario búlgaro**, una baraja de N cartas se divide en pilas con n_i cartas cada una, de modo que $\sum n_i = N$. En cada paso, se retira una carta de cada pila; las cartas retiradas forman una nueva pila. Las pilas vacías se eliminan.

Si $N = T_k$ entonces el juego siempre alcanza la configuración estable $\{1, 2, ..., k\}$ en un máximo de $k^2 - k$ movimientos.

Para $N = 15$ con pilas iniciales $\{5, 7, 3\}$:

```
5 7 3                                                    Display
4 6 2 3
3 5 1 2 4
2 4 1 3 5
1 3 2 4 5
2 1 3 4 5
1 2 3 4 5
```

Diseña un programa usando una estructura para almacenar el número de pilas y sus tamaños. Para el ejemplo anterior, el estado inicial sería `{3, {5, 7, 3}}`, con soporte para hasta `MAX=20` pilas. Implementa procedimientos para realizar un paso, imprimir el estado actual y comprobar si se ha alcanzado el estado estable. El programa imprime cada configuración hasta alcanzar la estabilidad, tal como se muestra. › Sol

o O o

Part 5 Manejo de ficheros

Contenido de esta parte Contenido del libro

5·1 Declaración de un manejador para manipular un fichero 203
5·2 Apertura y cierre del fichero 203
5·3 Lectura y escritura de un fichero de texto 204
5·4 Los flujos como parámetros de subprogramas 206



Los ficheros son esenciales para cualquier programa cuyos datos deban sobrevivir a una sola ejecución, es decir, para casi cualquier programa útil.

Un fichero es un repositorio de información almacenada en un dispositivo externo, separado de la memoria principal. Los ficheros son persistentes: permanecen intactos incluso después de apagar el sistema.

Existen dos tipos principales de ficheros: binarios y de texto. Los ficheros de texto están diseñados para ser leídos por personas y pueden contener marcado legible, como `<bold>This is bold</bold>`. Los ficheros binarios, en cambio, dependen de programas específicos que entienden su estructura. Con el tiempo, la mayoría de los formatos binarios —con la excepción de estándares bien establecidos como `.pdf` o `.jpg`— corren el riesgo de quedar obsoletos si no se mantiene el software capaz de leerlos y manipularlos, lo que con frecuencia resulta inviable. En consecuencia, los ficheros de texto siguen siendo el único formato verdaderamente universal con perspectivas reales de preservación a largo plazo.

Hasta la era de internet, las extensiones de los ficheros eran la principal forma de determinar el formato de un fichero. Esto sigue siendo válido, pero la transmisión de ficheros como flujos de bytes —por ejemplo, enlazados desde `.html`— hizo necesario añadirles metadatos (el tipo MIME) para informar al navegador qué clase de fichero está descargando, permitiéndole manejar el contenido de forma adecuada.

Los ficheros se identifican dentro de cada directorio únicamente por su nombre, por lo que cada nombre debe ser único dentro de ese directorio.

Las extensiones de fichero sirven para indicar el formato del contenido y/o qué programa puede manejarlo.

- `.cpp` Fichero de código fuente C++ en texto plano
- `.h`, `.hpp` Fichero de cabecera C++ en texto plano
- `.html` Fichero de Lenguaje de Marcado de Hipertexto
- `.pdf` Documento binario con texto formateado, imágenes, etc.
- `.jpg`, `.gif`, etc. Imágenes binarias
- `.zip` Archivo comprimido de uno o más ficheros
- `.txt` Fichero de texto

Trabajar con ficheros en todos los lenguajes de programación requiere los siguientes cuatro pasos:

1. Declarar el fichero.

2. Abrir el fichero.
   a. Si el fichero no existe todavía e intentamos abrirlo para lectura, se producirá un error.
   b. Si el fichero no existe todavía y lo abrimos para escritura, se creará vacío.
   c. Si el fichero ya existe y lo abrimos para escritura, su contenido anterior se borrará y quedará vacío.
   d. Si lo abrimos en modo *añadir* (*append*), el fichero se crea si no existe; si existe, su contenido se conserva y todo lo que escribamos va al final.
   e. También podemos abrirlo para lectura y escritura simultáneas. En ese caso el fichero debe existir ya y su contenido se conserva, pero podemos mover el cursor y sobreescribir partes concretas; esto se hace principalmente con ficheros binarios.
3. Leer del fichero y/o escribir en él.
4. Cerrar el fichero.

5·1 Declaración de un manejador para manipular un fichero

En C++ existen tres tipos principales de flujos de fichero (otras construcciones, como los flujos para canales de comunicación, se tratan de forma similar, pero no se abordan aquí).

- `ifstream`: flujo de fichero de *entrada* (*input*), para leer ficheros existentes.
- `ofstream`: flujo de fichero de *salida* (*output*), para crear ficheros y escribir en ellos.
- `fstream`: flujo de fichero para lectura y escritura simultáneas.

5·2 Apertura y cierre del fichero

El caso más sencillo es abrir un fichero existente para lectura en modo texto:

```C++
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in;

  in.open("customer.txt");
  if (not in) {
    cerr << "Error opening customer.txt" << endl;
  } else {
    // ... read from in ...
    in.close();
  }
  return 0;
}
```

Obsérvese que necesitamos la cabecera `<fstream>`, que contiene las declaraciones necesarias para manejar ficheros en C++.

Podemos declarar y abrir el fichero en un solo paso, pasando su nombre al constructor:

- `ifstream in    (filename);`
- `ofstream out   (filename);`
- `fstream  inout (filename, ios::in | ios::out);`

Tanto `open` como estos constructores aceptan un segundo parámetro opcional, el *modo de apertura*. `ifstream` siempre añade `ios::in`, `ofstream` siempre añade `ios::out`, y `fstream` utiliza por defecto `ios::in | ios::out`. Se pueden combinar varios indicadores con `|`:

- `ios::in` Modo entrada. La información se lee del fichero.
- `ios::out` Modo salida. La información se escribe en el fichero. Por sí solo, vacía un fichero existente (implica `ios::trunc`).
- `ios::app` Modo añadir (*append*). Si el fichero ya existe, su contenido se preserva y toda la salida se escribe al final del fichero.
- `ios::ate` (*at end*, al final) El cursor se sitúa al final del fichero justo tras abrirlo; a diferencia de `app`, puede desplazarse posteriormente.
- `ios::trunc` Si el fichero existe, su contenido se elimina.
- `ios::binary` Modo binario. Los bytes se transfieren exactamente como están, sin traducir los finales de línea (lo que importa en Windows).
- `ios::noreplace` (C++23, solo con `out`) Si el fichero ya existe, la apertura falla, con lo que no puede sobreescribirse por accidente.

Los ficheros son susceptibles al problema habitual de no existir o ser ilegibles cuando se espera que estén disponibles. Por tanto, antes de proceder con las operaciones de lectura, la primera y más importante comprobación que debemos realizar es si el fichero se abrió correctamente.

```
  in.open("customer.txt");
  if (not in)
  if (in.fail())
  if (not in.is_open())
```

Estas son tres formas de verificar que el fichero se abrió correctamente.

Un nombre de fichero sin directorio, como `"customer.txt"`, es relativo al *directorio de trabajo* del programa: el directorio desde el que se ejecuta el programa, que no es necesariamente donde están el código fuente o el ejecutable. Los entornos de desarrollo (IDEs) suelen ejecutar los programas desde una carpeta diferente, y esta es la causa más frecuente de errores de «fichero no encontrado». En caso de duda, ejecute el programa desde un terminal en el directorio donde se encuentra el fichero, o indique la ruta completa.

Cuando una aplicación termina, el sistema cierra todos los ficheros que la aplicación tenía abiertos, y un flujo también se cierra automáticamente cuando su variable sale del ámbito. No obstante, es buena práctica cerrar los ficheros explícitamente en cuanto ya no se necesiten: la salida se guarda en un *buffer* de memoria y solo se garantiza que llegue al disco cuando el fichero se cierra (o se vacía con *flush*); además, cerrarlo libera el fichero para otros programas. También puede ser necesario cerrar un fichero y reabrirlo más adelante, quizás en otro modo, dentro del mismo programa. Para ello se usa el método `.close()`:

```
in.close();
out.close();
```

5·3 Lectura y escritura de un fichero de texto

Hemos explorado varios métodos para abrir un fichero, pero ¿cómo trabajamos con su contenido?

Los ficheros no son como los arrays; no se puede acceder a su contenido mediante un índice. Al abrir un fichero, un marcador de posición llamado *cursor* indica dónde se producirá la siguiente lectura o escritura. Según el modo en que se abra el fichero, comienza al principio (cuando se abre para lectura o escritura) o al final (cuando se abre para añadir).

Tras cada operación de lectura o escritura, el cursor avanza en consecuencia.

La lectura y la escritura utilizan exactamente los mismos operadores y funciones que `cin` y `cout`: `<<`, `>>`, `getline`, etc. A continuación se muestra un programa sencillo que escribe dos nombres, uno por línea, cierra el fichero a modo de demostración y luego lo vuelve a abrir en modo *append* para añadir dos líneas más.

```C++
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  fstream f;
  f.open("demofile.txt", ios::out);             // created, or emptied
  if (not f) {
    cerr << "Cannot create demofile.txt" << endl;
  } else {
    f << "Juan" << endl;
    f << "Miguel" << endl;
    f.close();

    f.open("demofile.txt", ios::out | ios::app); // kept, write at the
    end
    f << "Paquito" << endl;
    f << "Vicenta" << endl;
    f.close();
  }
  return 0;
}
```

Para escribir un único carácter se usa `out.put(c)`; su contrapartida de lectura es `in.get(c)`, que lee un carácter incluyendo espacios y saltos de línea, a diferencia de `>>`. En ambos casos el cursor avanza una posición.

Al leer un fichero es necesario determinar cuándo detenerse. Los ficheros se leen secuencialmente y sabemos que hemos llegado al final porque una operación de lectura *falla*. Toda lectura —`in >> x`, `getline(in, s)`, `in.get(c)`— devuelve el propio flujo, que se evalúa como `true` si la lectura tuvo éxito y como `false` en caso contrario. De ahí el idioma: usar la propia lectura como condición del bucle, como en `while (in >> x)` o:

```C++
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  ifstream in("textfilewithlines.txt");
  if (not in) {
    cerr << "Cannot open textfilewithlines.txt" << endl;
  } else {
    string s;
    while (getline(in, s)) {
      cout << s << endl;           // display it
    }
```

```
    in.close();
  }
  return 0;
}
```

Un error muy frecuente es escribir `while (not in.eof())`. La función `eof()` solo se vuelve `true` *después* de que una lectura haya intentado ir más allá del final, por lo que el cuerpo del bucle se ejecuta una vez más con una lectura fallida, procesando típicamente el último elemento dos veces o procesando basura.

Si tenemos un fichero `people.txt` con una persona por línea, el nombre y la edad separados por `;`:

```
Juan Luis; 21                                                    Display
Pedro; 18
Juanito Maravillas; 14
```

el siguiente programa lo carga en un array:

```
#include <iostream>                                                    C++
#include <fstream>
#include <string>
#include <array>
using namespace std;

const int MAX = 200;
struct TPers {
  string name;
  int age;
};
typedef array<TPers,MAX> TPeople;

int main() {
  TPeople persons;
  int cnt = 0;                    // number of persons read

  ifstream in("people.txt");
  if (not in) {
    cerr << "Cannot open people.txt" << endl;
  } else {
    while (getline(in, persons[cnt].name, ';')) {
      in >> persons[cnt].age;     // reads the age
      in.ignore(100, '\n');       // discards the rest of the line
      cout << persons[cnt].name << " (" << persons[cnt].age << ")" <<
      endl;
      ++cnt;
    }
    in.close();
  }
  // persons[0] .. persons[cnt-1] hold the data of the file
  return 0;
}
```

Dos detalles merecen consideración. Primero, suponemos que la capacidad del array (`MAX`) es suficiente para acomodar el número de líneas del fichero. Segundo, `in.ignore(100, '\n')` es necesario porque el operador `>>` se detiene justo antes del carácter de salto de línea; sin él, ese `'\n'` se convertiría en el primer carácter del siguiente nombre.

5·4 Los flujos como parámetros de subprogramas

Las variables de flujo siempre deben pasarse a las funciones por referencia: no pueden copiarse, y cada lectura o escritura las modifica (su cursor y su estado de error).

```
bool openFileIn(ifstream& file, string name) {
  file.open(name);
  return file.is_open();
}
```

Mejor aún, declarar el parámetro como `ostream&` (o `istream&` para lectura). Tanto `cout` como cualquier `ofstream` son `ostream`, de modo que la misma función sirve para la pantalla y para un fichero:

```
void printPerson(ostream& out, TPers p) {
  out << p.name << "; " << p.age << endl;
}
```

y luego

```
  printPerson(cout, persons[0]);    // on the screen
  printPerson(outFile, persons[0]); // into a file
```

Obsérvese que `printPerson` escribe en el mismo formato que el programa anterior lee, por lo que un fichero guardado con ella puede cargarse de nuevo más adelante.

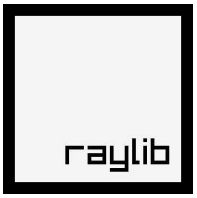
Part 6 Gráficos, sonido y WIMP con C++

Contenido de esta parte Contenido del libro

6·1 Preparación de las herramientas 208
6·2 **raylib**: dibujo, animación y sonido 210
6·3 **FLTK**: ventanas, botones y eventos 213

Hasta ahora todos los programas de este libro se han comunicado con el mundo a través de `cout` y `cin`: texto plano en un terminal. Este apéndice muestra cómo abrir una ventana real, dibujar en ella, animar, producir sonido y reaccionar al ratón. Nada de esto forma parte del C++ estándar —el lenguaje en sí no sabe nada de ventanas—, por lo que nos apoyamos en dos pequeñas bibliotecas gratuitas y multiplataforma.

119: **raylib** para gráficos, animación y sonido



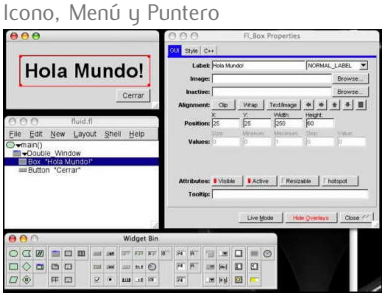
Usamos **raylib** para gráficos, animación y sonido, y **FLTK** para programas de ventanas tradicionales con botones y menús (lo que se conoce como interfaz *WIMP*: Windows, Icons, Menus, Pointer; es decir, Ventanas, Iconos, Menús, Puntero). Ambas son de código abierto, ambas funcionan en Windows, macOS y GNU/Linux, y ambas son lo suficientemente ligeras como para instalarse en pocos minutos.

120: FLTK para construir programas WIMP



raylib ofrece un lienzo en blanco y un bucle rápido — perfecto para juegos, dibujos y animaciones, pero no dispone de botones ni menús propios. **FLTK** proporciona los botones y los menús, pero no está pensado para dibujar juegos.

121: Una aplicación típica de Ventana, Icono, Menú y Puntero



6·1 Preparación de las herramientas

Solo hay que hacerlo una vez. Las instrucciones difieren según el sistema operativo, así que lea únicamente la sección que corresponda a su equipo.

Un consejo práctico antes de empezar: en macOS y GNU/Linux la forma más sencilla y fiable de compilar estos programas es el **Terminal**, usando un único comando que reutilizará siempre. En Windows, un entorno de desarrollo como **Code::Blocks** es el camino más cómodo, al menos para **raylib**. No luche contra su sistema; use la herramienta que le resulte más fácil.

6·1.1 En macOS

Primero, si aún no lo ha instalado, instale el compilador de línea de comandos de Apple (esto también le proporciona `c++`):

```
xcode-select --install
```

Luego instale las bibliotecas con **Homebrew** (`brew`):

```
brew install raylib
brew install fltk
```

Eso es todo. Pase directamente a *Cómo compilar y ejecutar*.

6·1.2 En GNU/Linux (Debian / Ubuntu)

Instale el compilador y las herramientas de construcción, y luego las bibliotecas:

```
sudo apt install build-essential
sudo apt install libraylib-dev
sudo apt install libfltk1.4-dev      # older systems: libfltk1.3-dev
```

Si su distribución no incluye `libraylib-dev`, siga la página wiki de **raylib** *«Working on GNU Linux»* para compilarla desde el código fuente una sola vez.

6·1.3 En Windows (Code::Blocks)

raylib. La vía más sencilla es el instalador propio de **raylib** para Windows, que incluye un compilador listo para usar (MinGW-w64) junto con la biblioteca. Tras instalarlo en `C:\raylib`:

1. En **Code::Blocks**, abra *Settings* → *Compiler* → *Toolchain executables* y apunte el directorio del compilador a `C:\raylib\MinGW`. *No* pulse «auto-detect».
2. Abra la plantilla de proyecto raylib que viene con el instalador o, en un proyecto nuevo, añada estas bibliotecas en la configuración del enlazador: `raylib`, `opengl32`, `gdi32`, `winmm`.

FLTK. **FLTK** en Windows requiere algo más de configuración que en macOS o Linux. El camino más accesible para un principiante es instalar **MSYS2**, añadir el paquete **FLTK** desde su shell (`pacman -S mingw-w64-ucrt-x86_64-fltk`) y luego compilar desde ese shell con exactamente el mismo comando `fltk-config` que se muestra a continuación para macOS y Linux. Su instructor también puede proporcionarle una configuración preconfigurada. Si **FLTK** le da problemas en Windows, realice los ejercicios de **FLTK** en un equipo Mac o Linux del laboratorio.

6·1.4 Cómo compilar y ejecutar

En macOS y GNU/Linux se compila desde el Terminal. El comando tiene una parte de aspecto inusual, `$(...)`, así que veámosla detenidamente.

Para un programa con **raylib**:

```
c++ myprogram.cpp    $(pkg-config --libs --cflags raylib)
./a.out
```

Para un programa con **FLTK**:

```
c++ myprogram.cpp    $(fltk-config --cxxflags --ldflags)
./a.out
```

La parte `$(...)` ejecuta un pequeño programa auxiliar (`pkg-config` para **raylib**, `fltk-config` para **FLTK**) que imprime todos los indicadores técnicos que necesita la biblioteca: dónde viven sus ficheros de cabecera y con qué bibliotecas enlazar. El shell inserta esa salida directamente en la línea de compilación. La clave es que **nunca hay que memorizar esos indicadores** —el auxiliar los conoce—. `-o myprogram` simplemente asigna nombre al fichero de salida; `./myprogram` lo ejecuta.

En Windows con **Code::Blocks** no se escribe esto: se pulsa *Build and run* (el engranaje con flecha verde), y la configuración del proyecto que se realizó antes proporciona la misma información.

6·2 raylib: dibujo, animación y sonido

Todo programa de **raylib** tiene el mismo esqueleto: abrir una ventana y luego ejecutar un bucle hasta que el usuario la cierre. En cada vuelta del bucle se borra la pantalla y se dibuja la imagen actual. Como se redibuja todo unas sesenta veces por segundo, mover una figura un poco en cada vuelta produce animación. Esto se llama *bucle de juego* (*game loop*), y *usted* es responsable de él.

6·2.1 Una ventana con un mensaje

Nuestro primer programa abre una ventana, dibuja un círculo (solo el contorno, es decir, una *circunferencia*) y escribe una palabra en su centro.

```C++
// raylibHello.cpp
// juanfc 2026-09-09
//
// brew install raylib
// c++ raylibHello.cpp $(pkg-config --libs --cflags raylib)

#include "raylib.h"
#include <string> // we are not including iostream, so..

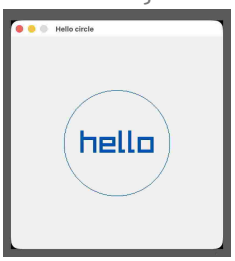
int main()
{
    InitWindow(400, 400, "Hello circle");
    SetTargetFPS(60);

    std::string message = "hello";
    int fontSize = 30;

    while (not WindowShouldClose()) {
        BeginDrawing();
            ClearBackground(RAYWHITE);
            DrawCircleLines(200, 200, 100.0f, DARKBLUE);
            DrawText(message.c_str(),
                     200 - MeasureText(message.c_str(), fontSize) / 2,
                     200 - fontSize / 2,
                     fontSize, DARKBLUE);
        EndDrawing();
    }

    CloseWindow();
    return 0;
}
```

122: La salida gráfica de un texto sencillo y un círculo con **raylib**



Tres aspectos a destacar:

1. `while (not WindowShouldClose())` es el bucle de juego. Solo termina cuando el usuario cierra la ventana o pulsa `Esc`.
2. **raylib** dibuja el texto desde su **esquina superior izquierda**, no desde su centro. Para centrar un mensaje de anchura w dentro de un espacio de anchura T, el margen izquierdo es $(T - w)/2$. Por eso restamos la mitad del ancho del texto (proporcionado por `MeasureText`) y la mitad del tamaño de fuente. Exactamente la misma idea centrará una ventana completa más adelante.
3. **raylib** es una biblioteca de C++ con interfaz C, por lo que `DrawText` espera un `const char *`, no un `std::string`. Se lo pasamos con `message.c_str()`. Este pequeño puente aparece siempre que se usa una biblioteca C desde C++.

6·2.2 Una animación: una pelota rebotante

Para animar, guardamos la posición y la velocidad de la pelota en un `struct`, y en cada vuelta del bucle la movemos y la hacemos rebotar contra las paredes. Agrupar los cinco números en una única `Ball` mantiene el programa ordenado: la pelota viaja de función en función como una sola entidad.

```cpp
// raylibBall.cpp
// juanfc 2026-09-09
//
// brew install raylib
// c++ raylibBall.cpp $(pkg-config --libs --cflags raylib)

#include "raylib.h"

struct Ball {
    float x;
    float y;
    float velocityX;
    float velocityY;
    float radius;
};

// Prototypes
void updateBall(Ball &ball, int width, int height);
void drawBall(Ball ball);

int main()
{
    const int screenWidth = 600;
    const int screenHeight = 400;

    InitWindow(screenWidth, screenHeight, "Bouncing ball");
    SetTargetFPS(60);

    Ball ball = {screenWidth / 2.0f, screenHeight /
                                2.0f, 4.0f, 3.0f, 20.0f};

    while (not WindowShouldClose()) {
        updateBall(ball, screenWidth, screenHeight);

        BeginDrawing();
            ClearBackground(RAYWHITE);
            drawBall(ball);
        EndDrawing();
    }

    CloseWindow();
    return 0;
}

void updateBall(Ball &ball, int width, int height)
{
    ball.x = ball.x + ball.velocityX;
    ball.y = ball.y + ball.velocityY;

    // Bounce off the walls by reversing the velocity component
    if (ball.x - ball.radius < 0 or ball.x + ball.radius > width) {
        ball.velocityX = -ball.velocityX;
    }
    if (ball.y - ball.radius < 0 or ball.y + ball.radius > height) {
        ball.velocityY = -ball.velocityY;
    }
}

void drawBall(Ball ball)
{
    DrawCircle(int(ball.x), int(ball.y), ball.radius, MAROON);
}
```

La física es la del instituto: cuando la pelota toca una pared, el componente de la velocidad perpendicular a esa pared cambia de signo.

123: Una animación sencilla de una pelota rebotando contra las paredes



Este ejemplo también muestra las dos formas de pasar un parámetro una al lado de la otra:

1. `updateBall(Ball &ball, ...)` recibe la pelota **por referencia** (`&`) porque debe **modificarla** —y esos cambios deben sobrevivir de vuelta en `main`.
2. `drawBall(Ball ball)` la recibe **por valor** (una copia) porque solo la **lee** para dibujar. Un refinamiento habitual en C++ es `const Ball &ball` para acceso de solo lectura: sin copia, y el compilador prohíbe cambios accidentales.

6·2.3 Producir sonido: una pequeña escala

Un sonido digital es simplemente una lista de números, las *muestras* (*samples*). Si producimos `44100` muestras por segundo siguiendo $\sin(2\pi ft)$, la tarjeta de sonido las convierte en un tono de frecuencia f. Cambiar f cambia la nota. Aquí construimos cada nota de la escala do-re-mi-fa-sol y la reproducimos.

```cpp
// raylibSound.cpp                                                    C++
// juanfc 2026-09-09
//
// brew install raylib
// c++ raylibSound.cpp $(pkg-config --libs --cflags raylib)

#include "raylib.h"
#include <array>
#include <cmath>
#include <thread>
#include <chrono>

// Prototypes
Sound makeNote(float frequency);

int main()
{
    InitAudioDevice();

    // do      re       mi       fa       sol   (A4 = 440 Hz)
    std::array<float, 5> scale = {261.63f, 293.66f, 329.63f, 349.23f,
    392.00f};

    for (float frequency : scale) {
        Sound note = makeNote(frequency);
        PlaySound(note);
        while (IsSoundPlaying(note)) {
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
        UnloadSound(note);
    }

    CloseAudioDevice();
    return 0;
}

Sound makeNote(float frequency)
{
    const int sampleRate = 44100;
    const int durationMs = 400;                          // 0.4 seconds
    const int frameCount = sampleRate * durationMs / 1000;

    // One sine-wave sample per frame, as 16-bit signed values
```

```
    std::array<short, frameCount> samples;
    for (int i = 0; i < frameCount; i = i + 1) {
        float t = float(i) / float(sampleRate);
        samples[i] = short(8000.0f * std::sin(2.0f * PI * frequency *
        t));
    }

    Wave wave;
    wave.frameCount = frameCount;
    wave.sampleRate = sampleRate;
    wave.sampleSize = 16;
    wave.channels   = 1;
    wave.data       = samples.data();

    Sound note = LoadSoundFromWave(wave);   // raylib copies the
    samples in
    return note;
}
```

Dos ideas en las que vale la pena detenerse.

La elección entre `for` y `while` no es decorativa aquí. Recorremos las cinco notas con un `for`, porque el número de vueltas se **conoce de antemano**. Esperamos a que una nota termine con un `while`, porque **no podemos saber** cuántas vueltas requerirá —depende del sonido—. Esta es exactamente la regla del capítulo sobre los bucles.

La espera usa la biblioteca estándar, `std::this_thread::sleep_for`, y **no** la función propia de **raylib** `WaitTime`. La razón es una trampa sutil: las funciones de temporización de **raylib** (`GetTime` y `WaitTime` por debajo) solo se ponen en marcha con `InitWindow`. Este programa no abre ninguna ventana, así que el reloj de **raylib** nunca arranca y `WaitTime` esperaría indefinidamente —se oiría una sola nota y el programa se bloquearía—. La función estándar `sleep_for` no depende del reloj de **raylib**, por lo que funciona tanto si hay una ventana como si no. La lección: hay que saber **qué inicializa cada llamada**. `InitAudioDevice` proporciona sonido, pero no el reloj.

1) ¿Por qué el bucle de las notas usa `for` mientras el bucle de «esperar hasta que termine» usa `while`? ¿Podrían intercambiarse?
2) Modifique `durationMs` para obtener notas más largas. ¿Qué línea hay que cambiar?
3) Añada dos notas más (la, si) para completar la octava. ¿Qué frecuencias necesita y hay que cambiar alguna otra línea?

6·3 FLTK: ventanas, botones y eventos

Los programas con **FLTK** tienen un aspecto diferente a los de **raylib**. No se escribe un bucle. En su lugar se construye la ventana, se indica a **FLTK** qué funciones propias debe llamar cuando ocurran cosas y luego se cede el control a **FLTK**. Esto es la programación **dirigida por eventos**.

6·3.1 Una ventana con un mensaje

```
#include <FL/Fl.H>                                                C++
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>
```

```
int main()
{
    Fl_Window *window = new Fl_Window(340, 180, "Hello");
    Fl_Box *box = new Fl_Box(20, 40, 300, 100, "Hello, world!");

    box->box(FL_UP_BOX);
    box->labelsize(28);
    box->labelfont(FL_BOLD);

    window->end();
    window->show();

    return Fl::run();
}
```

Tres puntos que sorprenden a los recién llegados:

1. El `new` aquí **no** es el descuidado `new` en bruto que evitamos en otros contextos. **FLTK** es propietaria de sus widgets: cada hijo (el cuadro) se adjunta a la ventana actual, y el destructor de la ventana elimina sus hijos. Crear un widget en la pila en su lugar haría que la ventana intentase eliminar un objeto de pila —lo que provocaría un fallo del programa (*crash*)—. En **FLTK**, `new` es el contrato.
2. `window->end()` significa «he terminado de añadir hijos a esta ventana». Todo lo creado entre el constructor de la ventana y `end()` se adjunta a ella automáticamente.
3. `Fl::run()` es el bucle de eventos de **FLTK**. Es lo que el bucle de juego era en **raylib**, pero aquí es la **biblioteca** quien lo ejecuta, no usted. Espera clics y pulsaciones de teclado y retorna únicamente cuando se cierra la ventana.

6·3.2 Un botón que reacciona

Para que ocurra algo al hacer clic registramos una **función de retorno de llamada** (*callback*): una función que **FLTK** llamará por nosotros cuando se pulse el botón. Guardamos todo lo que necesita la *callback* en un `struct`, de modo que seguimos sin usar variables globales. También centramos la ventana en la pantalla.

```
#include <FL/Fl.H>                                                    C++
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>
#include <FL/Fl_Button.H>
#include <cstdio>

struct Counter {
    Fl_Box *display;
    int clicks;
};

// Prototypes
void onClick(Fl_Widget *widget, void *data);

int main()
{
    Fl_Window *window = new Fl_Window(340, 180, "Click me");
    Fl_Box *box       = new Fl_Box(20, 30, 300, 60, "No clicks yet");
    Fl_Button *button = new Fl_Button(120, 110, 100, 40, "Click");

    box->box(FL_UP_BOX);
    box->labelsize(20);

    Counter counter = {box, 0};
    button->callback(onClick, &counter);
```

```
    window->end();
    window->position((Fl::w() - window->w()) / 2,
                     (Fl::h() - window->h()) / 2);
    window->show();

    return Fl::run();
}

void onClick(Fl_Widget *widget, void *data)
{
    Counter *counter = (Counter *)data;
    counter->clicks = counter->clicks + 1;

    char text[64];
    snprintf(text, sizeof(text), "Clicks: %d", counter->clicks);
    counter->display->copy_label(text);
}
```

La idea central es la ***callback***, y se entiende mejor por contraste con **raylib**. En **raylib usted** preguntaba en cada vuelta del bucle: «¿ha hecho algo el usuario?» —eso es el *sondeo* (*polling*)—. En **FLTK** deja su número de teléfono y **FLTK** le llama: registra `onClick`, y **FLTK** la invoca exactamente cuando se produce el clic. Ya no controla el bucle. Esta cesión se denomina *inversión de control*: «no nos llame usted, ya le llamaremos nosotros».

Algunos detalles que conectan con capítulos anteriores:

1. El `void *data` es la forma de C de pasar «sus propias cosas» a una función que no conoce su tipo. Ponemos todo lo que necesita la *callback* en un `struct`, pasamos su dirección y lo recuperamos dentro con el cast `(Counter *)data`. Obsérvese que `counter` vive en `main` durante toda la ejecución de `Fl::run()`, por lo que su dirección sigue siendo válida —sin puntero colgante (*dangling pointer*)—.
2. Llamamos a `copy_label` y no a `label` deliberadamente. `label` almacenaría solo un **puntero** a `text`, que es local y desaparece cuando `onClick` retorna —estaríamos apuntando a memoria muerta—. `copy_label` le pide a **FLTK** que conserve su **propia copia**. Siempre hay que preguntarse: ¿quién es el propietario de esta memoria?
3. Centrar la ventana reutiliza la fórmula del margen del ejemplo del círculo: `Fl::w()` y `Fl::h()` son el tamaño de la pantalla, `window->w()` y `window->h()` el tamaño de la ventana, y $(T - w)/2$ es el margen. Debe hacerse **antes** de `show()`, porque la ventana toma su posición cuando se muestra.

1) En `onClick`, ¿qué ocurriría si usásemos `label` en lugar de `copy_label`?

2) El parámetro `widget` no se usa aquí. ¿Qué le pasa **FLTK** en él y cuándo lo necesitaría?

3) Reescriba el centrado para que la ventana quede en la esquina **superior derecha** en lugar de en el centro.

Part 7 Estilo de programación

Contenido de esta parte Contenido del libro

7·1 Comentarios de grandes programadores 216
7·2 Estilo y requisitos 218

7·1 Comentarios de grandes programadores

Antes de las reglas, el espíritu. Así describen *el código limpio* varios de los maestros del campo.

Estas definiciones fueron recopiladas por Robert C. Martin en *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008. (Martin 2008)

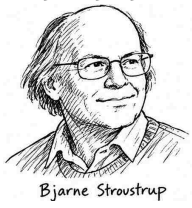
> Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.
>
> — Grady Booch

124: Autor de «Object-Oriented Analysis and Design with Applications»



> I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.
>
> — Bjarne Stroustrup

125: El creador de C++ y autor de *The C++ Programming Language*



> Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.
>
> — Dave Thomas

126: Fundador de OTI y arquitecto de la estrategia Eclipse



> I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. Clean code always looks like it was written by someone who cares.
>
> — Michael Feathers

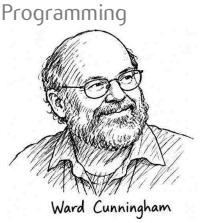
127: Autor de *Working Effectively with Legacy Code*



You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem.

— Ward Cunningham

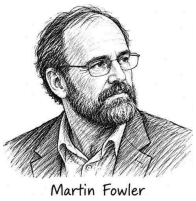
128: Inventor de Wiki y coinventor de eXtreme Programming



Any fool can write code that a computer can understand. Good programmers write code that humans can understand

— Martin Fowler

129: Desarrollador, 1963-



Referencias

MARTIN, Robert C., 2008. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. ISBN 9780132350884.

7·2 Estilo y requisitos

La tabla siguiente resume los puntos clave tratados a lo largo del curso; ◆ indica prerrequisitos:

		Estándares y requisitos de estilo (◆ Obligatorio)
1	◆	Nunca declarar variables globales fuera de los subprogramas
2	◆	No interrumpir ningún bucle con sentencias `return` o `break`
3	◆	Solo se permiten las estructuras de selección e iteración tratadas en los apuntes, transparencias y demostraciones de clase (`if`, `switch`, `while`, `do-`while«, `for`, etc.). Otras construcciones, como ~~`goto`~~ y ~~`continue`~~, están estrictamente prohibidas
4	◆	La condición booleana de la segunda sección de un bucle `for` solo debe depender de la variable de iteración, y la variable de iteración debe actualizarse exclusivamente en la tercera sección de la declaración del bucle
5	◆	No usar declaraciones de arrays al estilo C como `int a[…];`. Usar siempre `string` y `array<..>` de la STL. Todas las declaraciones de `array` deben hacerse con `typedef`
6		Prestar especial atención a la indentación y el formato del código, prefiriendo el estilo Kernighan/Ritchie

7		Esquema de la estructura de un programa:
		`#include // (from general to specific headers)`
		`const`
		`typedef`
		`int main()  // with prototypes(..) at the beginning of their body`
		`subprogramBodies() // from more abstract to more detailed { with prototypes(..) }`

8		Usar bucles `for` solo cuando se sabe de antemano, antes de que comience el bucle, cuántas iteraciones se realizarán; aplicarlos a rangos o secuencias con límites bien definidos. No incluir condiciones dentro del bucle que puedan cambiar durante la iteración
9		Usar bucles `while` para la iteración de propósito general, especialmente cuando la ejecución del bucle depende de los efectos secundarios del cuerpo del bucle
10		Usar nombres descriptivos para todos los identificadores, especialmente para las variables de gran ámbito o de importancia significativa en el programa (`finalLength`, `volume`, `avgPrice`, etc.)
11		Reservar nombres cortos estándar exclusivamente para variables de vida corta, como índices de bucle, contadores y construcciones similares (`i`, `s`, `n`, etc.)
12		Declarar las variables en sus ámbitos respectivos y situarlas cerca de su primer uso, especialmente en el caso de variables de vida corta. Evitar agrupar las declaraciones de variables al comienzo de los subprogramas
13		Los nombres de los nuevos tipos creados con `struct`, `enum` o `typedef` deben comenzar con la letra T (por ejemplo, `TList`, `TStudentIP`)
14		Declarar los prototipos de los subprogramas necesarios al comienzo de cada subprograma, antes de declarar las variables locales
15		Solo los subprogramas **diseñados para la interacción con el usuario** deben realizar operaciones de entrada/salida mediante `cin`/`cout`. Los subprogramas que procesan parámetros y devuelven resultados, o los subprogramas no destinados a la interacción con el usuario, no deben realizar operaciones de entrada/salida
16		Usar una única sentencia `return` solo al final de las funciones. Se permiten múltiples sentencias `return` únicamente dentro de una estructura `if`/`switch` final claramente definida

17 Usar las palabras `and` `or` `not` para combinar booleanos en lugar de los caracteres menos legibles `&&` `||` `!` (Recuerde que siempre puede usar `<` `>` `<=` `>` `!=` `==` en comparaciones)

o O o