UNIVERSIDAD DE GRANADA

Fundamentos de Programación

Temario de la asignatura

Ismael Sallami Moreno

Recursos Ingeniería Informática y Ade

Licencia

Fundamentos de Programación

Ismael Sallami Moreno

https://elblogdeismael.github.io

Índice general

Capítulo 1

Programación en C++: fundamentos

Tema 1 del programa. Qué es un algoritmo y qué lo distingue de un programa, cómo se representa la información dentro de la máquina, y los tipos, operadores y expresiones con los que se empieza a escribir en C++.

1.1 El ordenador, algoritmos y programas

Un **algoritmo** es una secuencia finita de pasos que resuelve un problema. Para serlo tiene que cumplir cinco propiedades:

Propiedad	Qué exige
Finitud	termina tras un número finito de pasos
Precisión	cada paso está definido sin ambigüedad
Entrada	recibe cero o más datos
Salida	produce al menos un resultado
Efectividad	cada paso es realizable en tiempo finito

Un **programa** es un algoritmo escrito en un lenguaje que la máquina puede ejecutar. La distinción importa porque el algoritmo se puede razonar, comparar y demostrar correcto con independencia del lenguaje; el programa añade los detalles que un lenguaje concreto exige.

1.1.1 Del código fuente al ejecutable

C++ es un lenguaje compilado, y el paso de un fichero de texto a un programa ejecutable tiene cuatro etapas:

Etapa	Herramienta	Entrada	Salida
Preprocesado	`cpp`	`.cpp`	texto expandido
Compilación	`g++`	texto expandido	ensamblador
Ensamblado	`as`	ensamblador	objeto `.o`
Enlazado	`ld`	objetos y bibliotecas	ejecutable

```
g++ -Wall -std=c++17 -o programa programa.cpp
./programa
```

La opción `-Wall` no es opcional en esta asignatura: activa los avisos que detectan variables sin inicializar, comparaciones sospechosas y valores de retorno olvidados. Un programa que compila sin avisos no es necesariamente correcto, pero uno que compila con avisos casi nunca lo es.

La alternativa a compilar es **interpretar**, que traduce y ejecuta a la vez. Compilar produce programas más rápidos y separa el momento de encontrar los errores del momento de ejecutar; interpretar acorta el ciclo de prueba. C++ está del lado compilado, y por eso el compilador detecta antes de ejecutar muchos errores que un lenguaje interpretado solo descubre al llegar a la línea.

1.1.2 Paradigmas

Paradigma	Idea central	Ejemplos
Imperativo	secuencia de instrucciones que cambian el estado	C, Pascal
Estructurado	imperativo con solo tres estructuras de control	el de esta asignatura
Orientado a objetos	datos y operaciones agrupados en clases	C++, Java
Funcional	composición de funciones sin estado mutable	Haskell, Lisp

C++ admite los tres primeros y parte del cuarto. La asignatura recorre el camino del estructurado al orientado a objetos: los cuatro primeros temas son programación estructurada y el tema 5 introduce las clases.

1.2 Especificación de programas

Antes de escribir código hay que decir **qué** hace el programa, no cómo. La especificación de una función se compone de tres partes:

Parte	Qué expresa
Precondición	lo que debe cumplirse al entrar
Postcondición	lo que se garantiza al salir
Descripción	qué hace, en una frase

```
/**
 * @brief Calcula la raiz cuadrada entera de un numero.
 * @param n Numero del que se calcula la raiz. Precondicion: n >= 0.
 * @return El mayor entero r tal que r*r <= n.
 */
int raizEntera(int n);
```

La precondición reparte la responsabilidad: si dice `n >= 0`, quien llama debe garantizarlo y la función no está obligada a comprobarlo. Escribirla evita el código defensivo repetido y, sobre todo, deja claro de quién es el error cuando algo falla.

Es el contenido de la práctica del seminario 3, y la razón de que la documentación se escriba **antes** que el cuerpo de la función.

1.3 Datos y tipos de datos

Un **tipo de dato** define un conjunto de valores y las operaciones válidas sobre ellos. Un **objeto** es una zona de memoria con un tipo y, casi siempre, un nombre. La distinción del enunciado de la asignatura: `int` es un tipo; `int x;` declara un objeto de ese tipo.

1.3.1 Representación de la información

Todo dato acaba siendo bits, y la codificación elegida determina el rango y la precisión.

Enteros sin signo. Con n bits se representan los valores de 0 a $2^n - 1$.

Enteros con signo, en complemento a dos. El rango es de -2^{n-1} a $2^{n-1} - 1$. Se usa esta codificación y no otra por dos razones: el cero tiene una única representación, y la resta se hace con el mismo circuito que la suma.

Con 8 bits:

Valor	Binario
127	`01111111`
1	`00000001`
0	`00000000`
−1	`11111111`
−128	`10000000`

El rango es asimétrico: hay un negativo más que positivos. De ahí que el opuesto del mínimo no sea representable, y que negar `INT_MIN` sea un error.

Reales, en coma flotante. Un número se guarda como signo, exponente y mantisa. La norma IEEE 754 fija dos tamaños:

Tipo	Bits	Dígitos decimales fiables	Rango aproximado
`float`	32	~7	$10^{\pm 38}$
`double`	64	~15	$10^{\pm 308}$

La consecuencia que hay que interiorizar desde el primer día: **no todos los decimales se representan exactamente**. `0.1` en binario es periódico, igual que un tercio en decimal, así que se guarda redondeado.

```
if (a == b)          // mal, con reales
if (fabs(a - b) < 1e-9)   // bien
```

Comparar reales con `==` es un error, no un estilo. `0.1 + 0.2 == 0.3` es falso.

Caracteres. Un `char` guarda un código numérico. ASCII usa 7 bits para 128 caracteres, y Unicode con codificación UTF-8 cubre el resto usando de uno a cuatro bytes. Que `'A'` valga 65 permite operar con caracteres como con enteros, que es lo que hace funcionar `c - '0'` para convertir un dígito.

1.3.2 Tipos comunes en C++

Tipo	Contenido	Tamaño típico
`bool`	`true` o `false`	1 byte
`char`	un carácter	1 byte
`int`	entero	4 bytes
`long long`	entero grande	8 bytes
`float`	real	4 bytes
`double`	real de doble precisión	8 bytes
`string`	cadena de caracteres	variable

`string` no es un tipo primitivo: es una clase de la biblioteca estándar, y por eso hace falta `#include <string>`. Se comporta como un tipo básico porque redefine los operadores, que es justo lo que el tema 5 explica.

1.3.3 Declaración e inicialización

```
int contador = 0;          // declarada e inicializada
const double PI = 3.14159; // constante: no se puede modificar
int sinValor;              // declarada sin inicializar: valor indeterminado
```

Leer una variable sin inicializar es **comportamiento indefinido**: el programa puede dar cualquier resultado, y a menudo da el correcto durante las pruebas y otro en la corrección. La regla es inicializar en la declaración, siempre.

Las constantes se declaran `const` y se nombran en mayúsculas. Un número suelto en medio del código —un *número mágico*— hay que ponerle nombre: `if (n > 100)` no dice nada, `if (n > MAX_ALUMNOS)` sí.

1.4 Operadores y expresiones

1.4.1 Aritméticos

Operador	Operación	Detalle
`+ - *`	suma, resta, producto	—
`/`	división	**entera si los dos operandos son enteros**
`%`	resto	solo enteros

La división es la trampa del tema:

```
int a = 7, b = 2;
cout << a / b;                          // 3, no 3.5
cout << static_cast<double>(a) / b;     // 3.5
cout << 1 / 2 * 4.0;                    // 0, porque 1/2 vale 0
```

`1 / 2 * 4.0` es 0 porque los operadores del mismo nivel se evalúan de izquierda a derecha: primero `1/2`, que es división entera.

1.4.2 Relacionales y lógicos

Operador	Significado
`== != < > <= >=`	comparaciones
`&&`	conjunción
`||`	disyunción
`!`	negación

`&&` y `||` evalúan **en cortocircuito**: si el primer operando decide el resultado, el segundo no se evalúa. No es una optimización, es semántica, y se usa deliberadamente:

```
if (i < n && v[i] > 0)      // correcto: no accede a v[i] si i >= n
if (v[i] > 0 && i < n)      // incorrecto: accede antes de comprobar
```

Y el error de escritura más frecuente del curso:

```
if (x = 5)    // asigna 5 y evalua a cierto: siempre entra
if (x == 5)   // compara
```

`-Wall` avisa de eso.

1.4.3 Asignación e incremento

```
x += 3;        // equivale a x = x + 3
x++;           // postincremento: usa el valor y luego incrementa
++x;           // preincremento: incrementa y luego usa
```

`a = b++` deja en `a` el valor **anterior** de `b`; `a = ++b`, el nuevo. Con objetos no primitivos el preincremento es además más eficiente, porque el postincremento tiene que guardar una copia del valor antiguo.

1.4.4 Precedencia y conversiones

De mayor a menor prioridad: `!`, luego `* / %`, luego `+ -`, luego los relacionales, luego `&&`, luego `||`, y por último la asignación. Ante la duda, paréntesis: son gratis y evitan errores.

En una expresión mixta, C++ **promociona** el tipo menor al mayor: `int` con `double` da `double`. Esa conversión implícita es cómoda y esconde pérdidas de información en la dirección contraria:

```
int n = 3.99;            // n vale 3: se trunca, no se redondea
double d = 5 / 2;        // d vale 2.0: la division fue entera
```

La conversión explícita se escribe `static_cast<double>(x)`, y no con la sintaxis heredada `(double)x`: la forma con plantilla es más fácil de localizar y el compilador comprueba que la conversión tenga sentido.

1.5 Entrada y salida

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

int main() {
    int edad;
    cout << "Introduzca su edad: ";
    cin >> edad;
    cout << "El ano que viene tendra " << edad + 1 << endl;
```

```
    return 0;
}
```

`cin >>` salta los espacios en blanco y se detiene en el primero que encuentra, así que no sirve para leer una línea con espacios. Para eso está `getline(cin, linea)`.

Y el problema clásico de mezclar los dos: `cin >> n` deja el salto de línea en el flujo, y el `getline` siguiente lee una cadena vacía. Se resuelve con `cin.ignore()` entre ambos.

Si la entrada no encaja con el tipo, el flujo entra en estado de error y las lecturas siguientes fallan sin bloquear. Comprobarlo es lo que distingue un programa robusto:

```
if (!(cin >> edad)) {
    cerr << "Entrada no valida" << endl;
    return 1;
}
```

`cerr` es el flujo de error, no almacenado en búfer, y es donde van los mensajes de error para no mezclarse con la salida del programa cuando esta se redirige.

1.6 Estructura de un programa

```
#include <iostream>        // directivas de preprocesado
#include <cmath>

using namespace std;

const double PI = 3.14159; // constantes globales

/**
 * @brief Calcula el area de un circulo.
 * @param radio Radio del circulo. Precondicion: radio >= 0.
 * @return El area.
 */
double area(double radio) {
    return PI * radio * radio;
}

int main() {
    double r;
    cout << "Radio: ";
    cin >> r;
    cout << "Area: " << area(r) << endl;
    return 0;
}
```

`main` devuelve `int`, y ese valor es el código de salida que el sistema recibe: 0 significa éxito. `return 0` al final es opcional en `main` y se escribe por claridad.

Sobre `using namespace std`: es cómodo en programas de aprendizaje y desaconsejado en código real, porque trae al ámbito global todos los nombres de la biblioteca estándar y provoca ambigüedades. En ficheros de cabecera no debe aparecer nunca. El desarrollo completo de estos fundamentos está en Garrido Carrillo, 2005, y el tratamiento del lenguaje en Stroustrup, 2015.

Capítulo 2

Estructuras de control

Tema 2 del programa. Las construcciones que deciden qué se ejecuta y cuántas veces: condicionales y repetitivas.

2.1 El teorema de la programación estructurada

Böhm y Jacopini demostraron en 1966 que **cualquier algoritmo se puede escribir con tres estructuras**: secuencia, selección e iteración. No hace falta nada más, y en particular no hace falta el salto incondicional.

Es el resultado que justifica el estilo de toda la asignatura. Un programa construido solo con esas tres tiene una propiedad valiosa: cada bloque tiene una entrada y una salida, así que se puede razonar sobre él por partes.

2.2 Estructura condicional

2.2.1 `if` simple y doble

```
if (nota >= 5) {
    cout << "Aprobado" << endl;
} else {
    cout << "Suspenso" << endl;
}
```

Las llaves son obligatorias por disciplina, aunque el cuerpo tenga una sola instrucción. Sin ellas, añadir una segunda línea más tarde produce un error que la indentación oculta:

```
if (x > 0)
    cout << "positivo";
    contador++;          // se ejecuta SIEMPRE: no esta dentro del if
```

2.2.2 Anidamiento y escalera

```
if (nota >= 9) {
    cout << "Sobresaliente";
} else if (nota >= 7) {
    cout << "Notable";
} else if (nota >= 5) {
    cout << "Aprobado";
} else {
    cout << "Suspenso";
}
```

El orden importa y no es intercambiable: si la primera comprobación fuera `nota >= 5`, todas las notas aprobadas caerían ahí y las demás ramas nunca se alcanzarían. En una escalera de rangos se comprueba **de más restrictivo a menos**.

Cada condición se evalúa sabiendo que las anteriores fallaron, así que `else if (nota >= 7)` significa en realidad «entre 7 y 9». Escribir `else if (nota >= 7 && nota < 9)` es redundante.

2.2.3 `switch`

```
switch (opcion) {
    case 1:
        alta();
        break;
    case 2:
    case 3:
        baja();
        break;
    default:
        cout << "Opcion no valida" << endl;
}
```

Tres reglas:

- **La expresión debe ser entera o de carácter.** No admite reales ni cadenas.
- **`break` es obligatorio** al final de cada caso, o la ejecución continúa por el siguiente. Ese comportamiento se llama caída, y es la fuente principal de errores con `switch`.
- **Dos casos seguidos sin código entre ellos** comparten el mismo tratamiento, y eso sí es un uso deliberado de la caída.

2.2.4 El operador condicional

```
int mayor = (a > b) ? a : b;
```

Es una expresión, no una instrucción, así que se puede usar donde hace falta un valor. Solo compensa cuando las dos alternativas son cortas; anidarlo produce código ilegible.

2.3 Estructuras repetitivas

2.3.1 `while`

Comprueba **antes** de ejecutar, así que el cuerpo puede no ejecutarse ninguna vez. Es la estructura adecuada cuando el número de repeticiones no se conoce de antemano.

```
int suma = 0, n;
cout << "Numeros (0 para terminar): ";
cin >> n;
while (n != 0) {
    suma += n;
    cin >> n;
}
cout << "Suma: " << suma << endl;
```

Ese esquema —leer antes del bucle y volver a leer al final del cuerpo— se llama **lectura adelantada**, y es el patrón estándar para procesar una secuencia terminada por un centinela.

Todo `while` correcto tiene tres piezas, y olvidar cualquiera lo rompe:

Pieza	Dónde va	Si falta
Inicialización	antes del bucle	la condición usa un valor indeterminado
Condición	en el `while`	—
Progreso hacia la condición	dentro del cuerpo	bucle infinito

2.3.2 do-while

Comprueba **después**, así que el cuerpo se ejecuta al menos una vez. Su uso natural es la validación de entrada:

```
int opcion;
do {
    cout << "Elija (1-3): ";
    cin >> opcion;
} while (opcion < 1 || opcion > 3);
```

El punto y coma final tras la condición es obligatorio, y olvidarlo es un error de compilación difícil de leer.

2.3.3 for

Reúne las tres piezas en una línea. Es la estructura para cuando el número de repeticiones se conoce:

```
for (int i = 0; i < n; i++) {
    cout << v[i] << " ";
}
```

La variable declarada en el `for` **existe solo dentro del bucle**. Eso es lo deseable: si hace falta después, es que el bucle no era el sitio para declararla.

Recorrer un vector desde `0` hasta `n-1` con `i < n` es el idioma correcto. Los dos errores clásicos:

```
for (int i = 0; i <= n; i++)   // accede a v[n]: fuera del vector
for (int i = 1; i < n; i++)    // se salta v[0]
```

Se llaman **errores de desplazamiento en uno**, y son los más frecuentes de la programación con vectores. En C++ no producen un error: leen o escriben memoria ajena, y el programa puede funcionar durante las pruebas.

2.3.4 Cuál usar

Situación	Estructura
Número de repeticiones conocido	`for`
Repetir mientras se cumpla una condición	`while`
Al menos una repetición	`do-while`

Las tres son intercambiables —cualquiera se escribe con cualquier otra—, así que elegir es una cuestión de que el código diga lo que hace.

2.3.5 Bucles anidados

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= i; j++) {
        cout << "*";
    }
    cout << endl;
}
```

El bucle interno se ejecuta completo por cada vuelta del externo, así que el número total de iteraciones es el producto. Cada nivel necesita su propia variable de control: reutilizar `i` en los dos rompe el externo.

2.3.6 `break` y `continue`

`break` sale del bucle; `continue` salta al final del cuerpo y sigue con la iteración siguiente.

Los dos rompen la propiedad de entrada única y salida única, así que se usan con moderación. `break` está justificado en una búsqueda que ya ha encontrado lo que buscaba; `continue`, casi nunca, porque suele ser un `if` mal escrito.

Un peligro concreto: en un `for`, `continue` **sí** ejecuta el incremento, pero en un `while` no ejecuta nada de lo que hubiera después. Un `continue` colocado antes de la línea que hace avanzar un `while` produce un bucle infinito.

2.4 Diseño y verificación de bucles

2.4.1 El invariante

Un **invariante** es una propiedad que se cumple antes y después de cada iteración. Formularlo es la forma de convencerse de que un bucle es correcto sin ejecutarlo.

```
int suma = 0;
for (int i = 0; i < n; i++) {
    suma += v[i];
}
// Invariante: al empezar cada vuelta, suma contiene v[0] + ... + v[i-1].
```

Al terminar, `i == n`, así que el invariante dice que `suma` contiene la suma de los `n` elementos. Eso es la demostración.

2.4.2 Terminación

Un bucle termina si alguna magnitud decrece hacia la condición de salida y no puede hacerlo indefinidamente. Los bucles infinitos accidentales tienen tres causas casi siempre:

Causa	Ejemplo
Falta el progreso	se olvida el `i++`
La condición no se puede alcanzar	`while (x != 0)` decrementando de dos en dos desde un impar
Se compara un real con `!=`	el redondeo hace que nunca coincida exactamente

2.4.3 Depuración

El seminario 2 de la asignatura trata esto, y las tres técnicas que valen:

1. **Trazar a mano** una ejecución corta, anotando el valor de cada variable en cada vuelta. Encuentra la mayoría de los errores de bucle.
2. **Imprimir** valores intermedios, en `cerr` para no mezclarlos con la salida.
3. **Usar el depurador**: punto de ruptura en el bucle, ejecución paso a paso y observación de las variables.

```
g++ -Wall -g -o prog prog.cpp
gdb ./prog
```

Orden de GDB	Efecto
`break 25`	punto de ruptura en la línea 25
`run`	ejecuta
`next`	siguiente línea, sin entrar en funciones
`step`	siguiente línea, entrando
`print x`	valor de una variable
`continue`	sigue hasta el próximo punto

`-g` incluye información de depuración. Sin ella, GDB no puede relacionar el código máquina con las líneas del fuente.

2.4.4 Casos de prueba

Un programa se prueba con casos elegidos, no con los que salgan. Los que hay que incluir siempre:

Caso	Por qué
Típico	comprueba la lógica normal
Vacío o cero repeticiones	el bucle puede no ejecutarse nunca
Un solo elemento	los casos límite del recorrido
Extremos del rango	el primero y el último
Entrada inválida	el programa no debe romperse

Las tres primeras filas encuentran la mayor parte de los errores de esta asignatura. El desarrollo de estas estructuras y de la metodología de diseño de bucles está en Garrido Carrillo, 2005 y en Savitch, 2017.

Capítulo 3

Funciones

Tema 3 del programa. Cómo se divide un programa en piezas, cómo se comunican esas piezas y cómo se diseñan para que sirvan de verdad.

3.1 Por qué dividir

Un programa de doscientas líneas en una sola función es difícil de leer, imposible de probar por partes y no reutilizable. Dividirlo en funciones da cuatro cosas:

Ventaja	En qué consiste
Abstracción	se usa la función sabiendo qué hace, sin saber cómo
Reutilización	se escribe una vez y se llama muchas
Prueba por partes	cada función se comprueba por separado
Legibilidad	el nombre de la función documenta la intención

La abstracción es la fundamental: quien llama a `ordenar(v, n)` no necesita saber si por dentro hay una ordenación por selección o por inserción.

3.2 Fundamentos

3.2.1 Declaración, definición y llamada

```
// Declaracion, o prototipo: dice como se llama.
double area(double base, double altura);

int main() {
    cout << area(3.0, 4.0) << endl;   // llamada
    return 0;
}

// Definicion: dice que hace.
double area(double base, double altura) {
    return base * altura / 2.0;
}
```

Separar prototipo y definición permite llamar a una función antes de escribirla, y es lo que hace posible que dos funciones se llamen mutuamente. El prototipo va antes de `main`; la definición, después.

3.2.2 Parámetros y argumentos

El **parámetro** es el nombre de la declaración; el **argumento** es el valor concreto de la llamada. En `area(3.0, 4.0)`, `base` y `altura` son parámetros; `3.0` y `4.0`, argumentos.

3.2.3 Paso por valor

El parámetro recibe una **copia** del argumento. Modificarlo dentro no afecta a quien llamó.

```
void incrementar(int x) {
    x++;                // modifica la copia
}

int n = 5;
incrementar(n);
cout << n;              // sigue valiendo 5
```

Es el paso por omisión, y el adecuado cuando la función solo necesita leer el valor.

3.2.4 Paso por referencia

El parámetro es **otro nombre** para el objeto del llamante. Se marca con `&`, y modificarlo sí afecta fuera.

```
void incrementar(int &x) {
    x++;                // modifica el original
}

int n = 5;
incrementar(n);
cout << n;              // ahora vale 6
```

Se usa en dos situaciones distintas:

1. **Cuando la función debe modificar el argumento**, que es la razón principal.
2. **Cuando el argumento es grande y copiarlo cuesta**. En ese caso se pasa por referencia **constante**, que evita la copia sin permitir modificar:

```
void imprimir(const string &texto);   // no copia y no modifica
```

3.2.5 Cuál elegir

Situación	Paso
Tipo básico que solo se lee	por valor
Objeto grande que solo se lee	por referencia constante
El argumento debe modificarse	por referencia
Devolver varios resultados	por referencia, o mejor un registro

La referencia constante es la opción por omisión para objetos: no copia y el compilador impide modificarlos, así que no hay que confiar en la disciplina.

3.2.6 Devolución de valores

Una función devuelve un valor con `return`, y `void` indica que no devuelve nada.

```
bool esPrimo(int n) {
    if (n < 2) return false;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return false;
    }
    return true;
}
```

`i * i <= n` en vez de `i <= n/2` no es una micro-optimización caprichosa: reduce las vueltas de $n/2$ a $\sqrt{n}$, que para un número de nueve cifras es la diferencia entre 500 millones de vueltas y 31 mil.

Todo camino debe devolver un valor. Una función no `void` que termina sin `return` produce comportamiento indefinido, y `-Wall` avisa.

3.2.7 La pila de llamadas

Cada llamada crea un **marco de activación** en la pila, con los parámetros, las variables locales y la dirección de retorno. Al volver, el marco desaparece.

```
+----------------------+
| marco de esPrimo     |  <- cima, la llamada actual
+----------------------+
| marco de main        |
+----------------------+
```

De ahí salen tres hechos:

- **Las variables locales de cada llamada son independientes.** Es lo que hace funcionar la recursión del tema 6.
- **Una variable local deja de existir al volver.** Devolver una referencia o un puntero a una local es un error: apunta a memoria que se reutilizará.
- **La pila es finita.** Una recursión sin caso base la agota, y el programa aborta con desbordamiento de pila.

3.2.8 Ámbito y vida

Clase	Dónde se ve	Cuánto vive
Local	dentro del bloque donde se declara	hasta que el bloque termina
Parámetro	dentro de la función	hasta que la función vuelve
Global	en todo el fichero desde su declaración	toda la ejecución
Estática local	dentro de la función	toda la ejecución

Una variable local que oculta a una global con el mismo nombre es legal y confuso. Y las **variables globales se evitan**: cualquier función puede modificarlas, así que un error puede estar en cualquier parte del programa. Las constantes globales sí son correctas, porque nadie las modifica.

3.3 Diseño de funciones

3.3.1 Descomposición descendente

Partir del problema completo y dividirlo en subproblemas hasta que cada uno quepa en una función:

```
gestionar notas
 +-- leer notas
 +-- calcular media
 +-- contar aprobados
 +-- mostrar informe
```

El enfoque contrario, **ascendente**, construye primero las piezas básicas y las combina. En la práctica se usan los dos: se planifica descendiendo y se escribe y prueba ascendiendo, empezando por las funciones que no dependen de nadie.

3.3.2 Qué hace una buena función

Criterio	Qué significa
Una responsabilidad	hace una cosa, y el nombre la dice entera
Nombre verbal y descriptivo	`calcularMedia`, no `proceso` ni `f1`
Pocos parámetros	más de cuatro o cinco suele indicar que faltaba un registro
Sin efectos ocultos	si modifica algo, se ve en su firma
Corta	si no cabe en una pantalla, probablemente hace dos cosas

El criterio de la responsabilidad única es el que más rinde. Una función llamada `leerYValidarYGuardar` está anunciando que debería ser tres.

3.3.3 Especificación

Ya apareció en el tema 1, y aquí es donde se aplica. Se escribe **antes** del cuerpo:

```
/**
 * @brief Busca un valor en un vector ordenado.
 * @param v Vector donde buscar. Precondicion: v esta ordenado
 *          de forma creciente.
 * @param n Numero de elementos. Precondicion: n >= 0.
 * @param x Valor buscado.
 * @return La posicion de x, o -1 si no esta.
 */
int buscar(const int v[], int n, int x);
```

La precondición «v está ordenado» es la que hace correcta la búsqueda dicotómica del tema 4. Sin escribirla, nada impide llamar a la función con un vector desordenado y obtener un resultado incorrecto sin que nada falle.

3.3.4 Sobrecarga

Varias funciones pueden compartir nombre si difieren en los parámetros:

```
double maximo(double a, double b);
int maximo(int a, int b);
int maximo(const int v[], int n);
```

El compilador elige según los argumentos. **No basta con que difiera el tipo de retorno**: la llamada no lo determina, así que dos funciones que solo se diferencien en eso son un error de compilación.

3.3.5 Parámetros con valor por omisión

```
void mostrar(double x, int decimales = 2);

mostrar(3.14159);      // usa 2
mostrar(3.14159, 4);   // usa 4
```

Los parámetros con valor por omisión deben ir **al final** de la lista, y el valor se escribe en el prototipo, no en la definición.

3.3.6 Modularización en ficheros

Cuando el programa crece, las funciones se reparten en ficheros:

```
// figuras.h
#ifndef FIGURAS_H
#define FIGURAS_H
double areaCirculo(double radio);
#endif
```

```
// figuras.cpp
#include "figuras.h"
const double PI = 3.14159265358979;
double areaCirculo(double radio) { return PI * radio * radio; }
```

```
g++ -Wall -c figuras.cpp
g++ -Wall -c principal.cpp
g++ figuras.o principal.o -o programa
```

El **guardián de inclusión** —las tres líneas del preprocesador— evita que la cabecera se procese dos veces cuando varios ficheros la incluyen. Sin él, las declaraciones se duplican y el compilador falla.

La cabecera lleva las declaraciones; el `.cpp`, las definiciones. Y en una cabecera no se escribe `using namespace std`, porque se lo impone a todo el que la incluya.

3.3.7 Comprobar que funciona

```
void probarBuscar() {
    int v[] = {1, 3, 5, 7, 9};
    assert(buscar(v, 5, 5) == 2);    // esta, en el medio
    assert(buscar(v, 5, 1) == 0);    // el primero
    assert(buscar(v, 5, 9) == 4);    // el ultimo
    assert(buscar(v, 5, 4) == -1);   // no esta
    assert(buscar(v, 0, 4) == -1);   // vector vacio
}
```

`assert` de `<cassert>` aborta si la condición es falsa, y desaparece al compilar con `-DNDEBUG`. Los

cinco casos son los del tema 2: típico, primero, último, ausente y vacío. El diseño modular y su metodología están desarrollados en Garrido Carrillo, 2016b, y los criterios de calidad de una función en Martin, 2008 y McConnell, 2004.

Capítulo 4

Registros, vectores y matrices

Tema 4 del programa. Los tipos que agrupan varios valores, y los algoritmos de búsqueda y ordenación que se construyen sobre ellos.

4.1 Registros

Un registro agrupa datos **de tipos distintos** bajo un nombre. En C++ se declara con `struct`:

```
struct Alumno {
    string nombre;
    int    edad;
    double nota;
};

Alumno a;
a.nombre = "Ana";
a.edad   = 19;
a.nota   = 8.5;
```

A los campos se accede con el punto. Un registro se puede asignar entero, pasar a una función y devolver:

```
Alumno leerAlumno() {
    Alumno a;
    cin >> a.nombre >> a.edad >> a.nota;
    return a;
}
```

Esa última propiedad resuelve el problema de devolver varios valores: en vez de tres parámetros por referencia, se devuelve un registro.

Los registros se anidan, y así se modela la estructura real de los datos:

```
struct Fecha  { int dia, mes, anio; };
struct Persona {
    string nombre;
    Fecha  nacimiento;
};

Persona p;
p.nacimiento.anio = 2005;
```

Un registro se pasa a una función **por referencia constante** salvo que haya que modificarlo: copiarlo entero cuesta, y con `const` el compilador garantiza que no se toca.

4.2 Vectores

Un vector guarda varios elementos **del mismo tipo** en posiciones consecutivas.

```
const int MAX = 100;
int v[MAX];

for (int i = 0; i < MAX; i++) {
    v[i] = 0;
}
```

Cuatro hechos que gobiernan todo el uso de vectores en C++:

1. **Los índices van de `0` a `n-1`.** El primero es `v[0]`.
2. **El tamaño es constante y se fija al compilar.** No se puede usar una variable leída del teclado como tamaño de un vector estático.
3. **No se comprueban los límites.** `v[MAX]` compila y accede a memoria ajena. El programa puede no fallar, y ese es el problema.
4. **No se pueden asignar ni comparar enteros.** `v = w` no copia el vector, y `v == w` compara direcciones. Hay que recorrer elemento a elemento.

4.2.1 Vector y tamaño lógico

Como el tamaño físico se fija al compilar y rara vez se llena, se lleva aparte cuántos elementos hay de verdad:

```
const int MAX = 100;
int  v[MAX];
int  n = 0;          // tamano logico: cuantos hay usados

// Anadir al final
if (n < MAX) {
    v[n] = x;
    n++;
}
```

El par vector-tamaño viaja siempre junto, y de ahí que las funciones reciban los dos parámetros. Agruparlos en un registro es la mejora natural, y es lo que el tema 5 formaliza como clase.

4.2.2 Vectores y funciones

Un vector **siempre se pasa por referencia**, aunque no se escriba `&`: lo que se pasa es la dirección del primer elemento. Por eso una función puede modificarlo, y por eso hay que pasar el tamaño aparte.

```
void imprimir(const int v[], int n);   // const: no lo modifica
void rellenar(int v[], int n);         // sin const: si lo modifica
```

Poner `const` cuando la función no modifica no es un adorno: documenta la intención y el compilador la comprueba.

4.2.3 Cadenas

`string` es un vector de caracteres con operaciones propias, y se comporta como un tipo básico porque la clase redefine los operadores:

```
string s = "Hola";
s += " mundo";           // concatena
cout << s.length();      // 10
cout << s[0];            // 'H'
cout << s.substr(0, 4);  // "Hola"
if (s == "Hola mundo") { /* compara contenido */ }
```

La diferencia con un vector clásico es exactamente esa: `==` compara el contenido y `=` copia. Es lo que las clases permiten hacer, y el tema 5 lo explica.

4.3 Algoritmos de búsqueda

4.3.1 Búsqueda lineal

Recorre el vector hasta encontrar el valor:

```
int buscarLineal(const int v[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (v[i] == x) return i;
    }
    return -1;
}
```

- No exige que el vector esté ordenado.
- Coste: $O(n)$ en el peor caso y en el medio, $O(1)$ en el mejor.

4.3.2 Búsqueda binaria

Sobre un vector **ordenado**, compara con el elemento central y descarta la mitad:

```
int buscarBinaria(const int v[], int n, int x) {
    int izq = 0, der = n - 1;
    while (izq <= der) {
        int centro = izq + (der - izq) / 2;
        if (v[centro] == x)      return centro;
        else if (v[centro] < x)  izq = centro + 1;
        else                     der = centro - 1;
    }
    return -1;
}
```

- **Precondición: el vector está ordenado.** Sin ella el resultado es incorrecto y nada avisa.
- Coste: $O(\log n)$.

Dos detalles del código que no son casuales:

- `izq + (der - izq) / 2` en vez de `(izq + der) / 2`. La segunda forma desborda cuando los dos índices son grandes, y es un error que estuvo presente durante veinte años en implementaciones muy usadas.
- La condición es `izq <= der`, con igual. Con `<` se pierde el caso en que queda un solo elemento por examinar.

La diferencia de coste es la que justifica ordenar: con un millón de elementos, la lineal examina un millón en el peor caso y la binaria veinte.

n	Lineal	Binaria
100	100	7
10 000	10 000	14
1 000 000	1 000 000	20

4.4 Algoritmos de ordenación

4.4.1 Selección

Busca el mínimo del resto y lo coloca en su sitio:

```
void ordenarSeleccion(int v[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int pos = i;
        for (int j = i + 1; j < n; j++) {
            if (v[j] < v[pos]) pos = j;
        }
        swap(v[i], v[pos]);
    }
}
```

Hace siempre $n(n-1)/2$ comparaciones, con independencia de los datos, y como mucho $n-1$ intercambios. Es la opción cuando mover elementos cuesta mucho.

4.4.2 Inserción

Coloca cada elemento en su posición dentro de la parte ya ordenada:

```
void ordenarInsercion(int v[], int n) {
    for (int i = 1; i < n; i++) {
        int actual = v[i];
        int j = i - 1;
        while (j >= 0 && v[j] > actual) {
            v[j + 1] = v[j];
            j--;
        }
        v[j + 1] = actual;
    }
}
```

Su ventaja está en el mejor caso: con el vector ya ordenado hace $n-1$ comparaciones y ningún desplazamiento, es decir $O(n)$. Por eso es la mejor de las tres para vectores casi ordenados, y por eso las bibliotecas la usan para los tramos pequeños dentro de algoritmos más elaborados.

4.4.3 Burbuja

Intercambia elementos adyacentes desordenados en pasadas sucesivas:

```
void ordenarBurbuja(int v[], int n) {
    bool cambio = true;
    for (int i = 0; i < n - 1 && cambio; i++) {
        cambio = false;
```

```
        for (int j = 0; j < n - 1 - i; j++) {
            if (v[j] > v[j + 1]) {
                swap(v[j], v[j + 1]);
                cambio = true;
            }
        }
    }
}
```

La bandera `cambio` es lo que la hace terminar antes si el vector ya está ordenado; sin ella, la burbuja no tiene ninguna ventaja sobre las otras dos.

4.4.4 Comparación

Algoritmo	Mejor	Medio	Peor	Estable	Intercambios
Selección	$O(n^2)$	$O(n^2)$	$O(n^2)$	no	$O(n)$
Inserción	$O(n)$	$O(n^2)$	$O(n^2)$	sí	$O(n^2)$
Burbuja	$O(n)$	$O(n^2)$	$O(n^2)$	sí	$O(n^2)$

Estable significa que dos elementos con la misma clave conservan su orden relativo. Importa cuando se ordena por un campo y ya estaba ordenado por otro: ordenar por apellido un listado ordenado por nombre deja, si el algoritmo es estable, los apellidos iguales ordenados por nombre.

Los tres son cuadráticos, y por eso no se usan en la práctica con vectores grandes: `sort` de la biblioteca estándar es $O(n \log n)$. Se estudian porque son la base sobre la que se construyen los buenos, y porque su análisis es el primer contacto con el coste de un algoritmo.

4.5 Matrices

Un vector de vectores. En C++ los elementos se guardan **por filas**:

```
const int F = 3, C = 4;
int m[F][C];

for (int i = 0; i < F; i++) {
    for (int j = 0; j < C; j++) {
        m[i][j] = i * C + j;
    }
}
```

La dirección de `m[i][j]` es $base + (i \cdot C + j) \cdot t$, con t el tamaño del elemento. De ahí sale la regla que en cursos posteriores se convierte en una cuestión de rendimiento: **recorrer por filas es más rápido que por columnas**, porque los elementos consecutivos de una fila están juntos en memoria.

Al pasar una matriz a una función hay que indicar todas las dimensiones salvo la primera, porque el compilador necesita el número de columnas para calcular la dirección:

```
void imprimir(const int m[][C], int filas);   // C debe ser constante conocida
```

Esa limitación es lo que hace incómodas las matrices estáticas, y la razón de que en la práctica se usen otras representaciones.

4.5.1 Operaciones habituales

```
// Suma de matrices
for (int i = 0; i < F; i++)
    for (int j = 0; j < C; j++)
        s[i][j] = a[i][j] + b[i][j];

// Transpuesta
for (int i = 0; i < F; i++)
    for (int j = 0; j < C; j++)
        t[j][i] = m[i][j];

// Producto: A es F x K, B es K x C
for (int i = 0; i < F; i++)
    for (int j = 0; j < C; j++) {
        p[i][j] = 0;
        for (int k = 0; k < K; k++)
            p[i][j] += a[i][k] * b[k][j];
    }
```

El producto es el primer algoritmo cúbico del curso: con matrices de 1000×1000 son mil millones de multiplicaciones. Es un buen sitio para ver que la complejidad no es una abstracción.

4.6 Vectores de registros

La combinación que aparece en casi todos los ejercicios:

```
const int MAX = 100;
Alumno clase[MAX];
int n = 0;

// Media de la clase
double suma = 0.0;
for (int i = 0; i < n; i++) suma += clase[i].nota;
double media = (n > 0) ? suma / n : 0.0;

// Ordenar por nota, de mayor a menor
for (int i = 0; i < n - 1; i++) {
    int pos = i;
    for (int j = i + 1; j < n; j++)
        if (clase[j].nota > clase[pos].nota) pos = j;
    swap(clase[i], clase[pos]);
}
```

La guarda `(n > 0)` antes de dividir no es paranoia: un vector vacío es un caso de prueba obligatorio, y sin ella el programa divide por cero. Los algoritmos de este tema, con su análisis, están desarrollados en Garrido Carrillo, 2005 y en Savitch, 2017.

Capítulo 5

Clases

Tema 5 del programa. El paso de la programación estructurada a la orientada a objetos: agrupar los datos con las operaciones que los manipulan, y ocultar cómo están representados.

5.1 De registro a clase

Un registro agrupa datos; una **clase** agrupa datos y las funciones que operan sobre ellos, y decide qué es visible desde fuera.

El problema que resuelve se ve con el vector del tema 4. Un vector y su tamaño lógico viajan siempre juntos, y nada obliga a mantenerlos coherentes: cualquier parte del programa puede escribir `n = 500` y romper el invariante. Con una clase, `n` es privado y solo las operaciones de la clase lo tocan.

```
class Fecha {
private:
    int dia_, mes_, anio_;

public:
    Fecha(int d, int m, int a);
    int  dia()  const;
    int  mes()  const;
    int  anio() const;
    void avanzarDia();
    bool esBisiesto() const;
};
```

Las convenciones que la asignatura usa: el guion bajo final en los datos privados para distinguirlos de los parámetros, y el orden privado antes que público.

5.2 Encapsulación

Los **datos miembro** guardan el estado; los **métodos** son las operaciones. Un método accede a los datos del objeto sobre el que se invoca sin recibirlos como parámetros.

```
Fecha hoy(14, 8, 2026);
hoy.avanzarDia();
cout << hoy.dia();
```

Dentro de un método, `dia_` se refiere al dato del objeto que recibió la llamada. El puntero implícito `this` apunta a ese objeto, y solo hace falta escribirlo cuando un parámetro oculta al dato miembro.

5.3 Ocultación de información

Ámbito	Quién accede
`private`	solo los métodos de la clase
`public`	cualquiera
`protected`	la clase y sus derivadas

La regla: **los datos privados, los métodos públicos que hagan falta**. Lo que gana:

- **El invariante se mantiene.** Si `mes_` solo se modifica por métodos que comprueban el rango, no puede valer 13.
- **La representación se puede cambiar.** Guardar la fecha como tres enteros o como días desde una referencia es una decisión interna; el código que usa la clase no cambia.
- **La superficie de error se reduce.** Un dato incorrecto solo puede haberlo escrito un método de la clase.

Los métodos que dan acceso a un dato se llaman **consultores** —o *getters*— y los que lo modifican, **modificadores**. Escribir mecánicamente un par por cada dato privado anula la encapsulación: si todo se puede leer y escribir desde fuera, la clase es un registro con más líneas. Se ofrecen las operaciones que el problema necesita, no accesos a los campos.

5.4 Constructores

Un constructor inicializa el objeto. Tiene el nombre de la clase, no devuelve nada y se ejecuta al crear el objeto.

```
class Fecha {
private:
    int dia_, mes_, anio_;

public:
    Fecha() : dia_(1), mes_(1), anio_(2000) {}          // por defecto

    Fecha(int d, int m, int a)                          // con parametros
        : dia_(d), mes_(m), anio_(a) {}
};
```

La lista después de los dos puntos es la **lista de inicialización**, y es la forma correcta de inicializar. La alternativa —asignar en el cuerpo— primero construye los miembros y después les asigna, así que hace el trabajo dos veces; y con miembros constantes o referencias no funciona en absoluto.

Si no se escribe ningún constructor, el compilador genera uno por defecto que **no inicializa los tipos básicos**. Un objeto recién creado tendría enteros con valor indeterminado, que es el error del tema 1 otra vez.

Y en cuanto se escribe un constructor con parámetros, el compilador deja de generar el de por defecto. Si hace falta, hay que escribirlo.

5.4.1 Validar en el constructor

```
Fecha::Fecha(int d, int m, int a) {
    if (m < 1 || m > 12) m = 1;
```

```
    if (d < 1 || d > diasDelMes(m, a)) d = 1;
    dia_ = d; mes_ = m; anio_ = a;
}
```

Comprobar aquí es lo que garantiza que **ningún objeto de la clase existe en un estado inválido**. Es el punto donde se establece el invariante que los demás métodos pueden dar por cierto.

5.5 Copias de objetos

5.5.1 Constructor de copia y asignación

```
Fecha a(14, 8, 2026);
Fecha b = a;      // constructor de copia
Fecha c;
c = a;            // operador de asignacion
```

El compilador genera los dos, y copian miembro a miembro. Con datos básicos eso basta.

Deja de bastar cuando la clase gestiona un recurso: si un miembro fuera un puntero a memoria reservada, la copia por omisión duplicaría el puntero y los dos objetos apuntarían al mismo sitio. Se llama **copia superficial**, y sus consecuencias son que modificar uno modifica el otro y que al destruirse ambos se libera dos veces la misma memoria.

En esa situación hay que escribir constructor de copia, operador de asignación y destructor. Es la regla de los tres, y aparece en cuanto la clase deja de contener solo tipos básicos.

5.5.2 Paso a funciones

Un objeto se pasa **por referencia constante**, salvo que la función deba modificarlo:

```
void mostrar(const Fecha &f);      // no copia, no modifica
void avanzar(Fecha &f);            // modifica
```

Pasarlo por valor invoca el constructor de copia, y con objetos grandes eso cuesta.

5.6 Datos y métodos constantes

Un método que no modifica el objeto se marca `const`:

```
int  dia()   const { return dia_; }
void avanzarDia();                  // sin const: si modifica
```

El compilador comprueba que un método `const` no modifica ningún dato, y **solo los métodos `const` se pueden invocar sobre un objeto constante**. De ahí que olvidar el `const` en un consultor rompa el código que recibe el objeto por referencia constante:

```
void mostrar(const Fecha &f) {
    cout << f.dia();     // error de compilacion si dia() no es const
}
```

Marcar `const` todo lo que no modifica es una disciplina que se paga sola: el compilador convierte en error de compilación lo que si no sería un error de lógica.

Un dato miembro `const` se inicializa en la lista de inicialización y no se puede modificar después. Y un miembro `static` es compartido por todos los objetos de la clase, no por cada uno: sirve para contadores de instancias y para constantes de la clase.

5.7 Sobrecarga de operadores

Una clase puede dar significado a los operadores del lenguaje:

```
class Fecha {
    // ...
public:
    bool operator==(const Fecha &otra) const;
    bool operator<(const Fecha &otra) const;
};

bool Fecha::operator<(const Fecha &otra) const {
    if (anio_ != otra.anio_) return anio_ < otra.anio_;
    if (mes_  != otra.mes_)  return mes_  < otra.mes_;
    return dia_ < otra.dia_;
}
```

Es lo que hace que `string` se comporte como un tipo básico: `s1 == s2` compara el contenido porque la clase define `operator==`.

La salida se sobrecarga con una función externa, porque el operando izquierdo es el flujo y no el objeto:

```
ostream & operator<<(ostream &os, const Fecha &f) {
    os << f.dia() << "/" << f.mes() << "/" << f.anio();
    return os;
}
```

Devolver el flujo por referencia es lo que permite encadenar: `cout << a << b`.

La regla al sobrecargar: **el operador debe significar lo que significa**. Redefinir `+` como una resta compila y hace el programa ilegible.

5.8 Colecciones: secuencia y tabla

La aplicación del tema. Una clase que encapsula el vector y su tamaño lógico, resolviendo el problema con el que empezó el capítulo:

```
class Secuencia {
private:
    static const int MAX = 100;
    int datos_[MAX];
    int n_;                       // invariante: 0 <= n_ <= MAX

public:
    Secuencia() : n_(0) {}

    int  tamanio() const { return n_; }
    bool vacia()   const { return n_ == 0; }
    bool llena()   const { return n_ == MAX; }

    bool anadir(int x) {
        if (llena()) return false;
```

```
        datos_[n_] = x;
        n_++;
        return true;
    }

    int elemento(int i) const {   // precondicion: 0 <= i < n_
        return datos_[i];
    }

    bool eliminar(int i) {
        if (i < 0 || i >= n_) return false;
        for (int j = i; j < n_ - 1; j++) datos_[j] = datos_[j + 1];
        n_--;
        return true;
    }
};
```

Lo que la clase garantiza y el vector suelto no:

- `n_` nunca supera `MAX`, porque `anadir` lo comprueba.
- `n_` nunca es negativo, porque `eliminar` valida el índice.
- Nadie de fuera puede escribir en `datos_` saltándose las comprobaciones.

La **tabla** es la otra colección del programa: guarda pares clave-valor y busca por clave.

```
struct Par { string clave; int valor; };

class Tabla {
private:
    static const int MAX = 100;
    Par datos_[MAX];
    int n_;

    int posicion(const string &c) const {   // privado: es un detalle interno
        for (int i = 0; i < n_; i++)
            if (datos_[i].clave == c) return i;
        return -1;
    }

public:
    Tabla() : n_(0) {}

    bool contiene(const string &c) const { return posicion(c) >= 0; }

    bool insertar(const string &c, int v) {
        int p = posicion(c);
        if (p >= 0) { datos_[p].valor = v; return true; }   // ya estaba
        if (n_ == MAX) return false;
        datos_[n_].clave = c;
        datos_[n_].valor = v;
        n_++;
```

```
        return true;
    }
};
```

`posicion` es privado a propósito: devuelve un índice del vector interno, que es justo lo que no se quiere exponer. Si se hiciera público, el código de fuera empezaría a depender de que la tabla es un vector, y cambiar la representación —por ejemplo a una tabla *hash*— dejaría de ser posible.

Esa es la idea central del tema: **la interfaz es lo que se promete, y la representación es lo que se puede cambiar**. El desarrollo de las clases en C++ está en Lafore, 2005 y en Garrido Carrillo, 2005, y las colecciones de la biblioteca estándar en Garrido Carrillo, 2016a.

Capítulo 6

Recursividad

Tema 6 del programa. Funciones que se llaman a sí mismas: cómo se diseñan, cómo se ejecutan por dentro y cuándo compensan frente a un bucle.

6.1 La idea

Una función es **recursiva** si se llama a sí misma, directa o indirectamente. Se apoya en que muchos problemas contienen versiones más pequeñas de sí mismos: el factorial de n es n por el factorial de $n-1$.

```
int factorial(int n) {
    if (n <= 1) return 1;          // caso base
    return n * factorial(n - 1);   // caso recursivo
}
```

6.2 Los tres requisitos

Un algoritmo recursivo correcto tiene tres piezas, y faltar cualquiera lo rompe:

Requisito	Qué exige	Si falta
Caso base	al menos un caso que se resuelve sin recursión	recursión infinita
Caso recursivo	resuelve el problema con una versión menor de sí mismo	no hay recursión
Convergencia	cada llamada se acerca al caso base	recursión infinita

La recursión infinita no cuelga el programa: **agota la pila** y aborta con desbordamiento. Es la señal inequívoca de que falta el caso base o de que las llamadas no convergen.

```
int mal(int n) {
    if (n == 0) return 1;
    return n * mal(n - 2);    // con n impar nunca llega a 0
}
```

Ese ejemplo tiene caso base y no converge: con `n` impar salta de 1 a -1, -3, y sigue indefinidamente. El caso base tiene que ser **alcanzable desde cualquier entrada válida**, y por eso `n <= 1` es mejor que `n == 1`.

6.3 Cómo se ejecuta

Cada llamada crea su propio marco en la pila, con sus parámetros y sus locales. Los marcos se apilan al descender y se desapilan al volver.

```
factorial(4)
 +-- 4 * factorial(3)
      +-- 3 * factorial(2)
           +-- 2 * factorial(1)
                +-- devuelve 1     <- caso base
           +-- devuelve 2
      +-- devuelve 6
 +-- devuelve 24
```

Dos consecuencias:

- **Cada llamada tiene sus propias variables.** Es lo que hace que la recursión funcione, y por qué en el tema 3 se insistía en que los marcos son independientes.
- **La profundidad está acotada.** La pila mide unos megabytes, así que una recursión de un millón de niveles la agota aunque el algoritmo sea correcto.

6.4 Diseño de algoritmos recursivos

El método, en cuatro pasos:

1. Identificar el caso base: la entrada más pequeña que se resuelve directamente.
2. Suponer que la función **ya funciona** para entradas menores. Es el paso difícil, y es exactamente la hipótesis de inducción.
3. Expresar el caso general en términos de esa suposición.
4. Comprobar que cada llamada se acerca al caso base.

El paso 2 es donde la gente se atasca: intentar seguir mentalmente todas las llamadas anidadas es imposible y no hace falta. Se confía en que la función resuelve el caso menor, igual que se confía en cualquier otra función ya escrita.

6.4.1 Ejemplos

Suma de un vector.

```
int suma(const int v[], int n) {
    if (n == 0) return 0;
    return v[n - 1] + suma(v, n - 1);
}
```

Potencia.

```
double potencia(double base, int exp) {
    if (exp == 0) return 1.0;
    if (exp < 0)  return 1.0 / potencia(base, -exp);
    return base * potencia(base, exp - 1);
}
```

Potencia rápida, que divide el exponente en vez de restarle uno:

```
double potenciaRapida(double base, int exp) {
    if (exp == 0) return 1.0;
```

```
    double mitad = potenciaRapida(base, exp / 2);
    if (exp % 2 == 0) return mitad * mitad;
    return base * mitad * mitad;
}
```

Pasa de $O(n)$ a $O(\log n)$, y la clave está en guardar el resultado en `mitad` en vez de llamar dos veces: llamar dos veces daría $O(n)$ otra vez, porque el número de llamadas se duplicaría en cada nivel.

Invertir una cadena.

```
string invertir(const string &s) {
    if (s.length() <= 1) return s;
    return invertir(s.substr(1)) + s[0];
}
```

Palíndromo.

```
bool esPalindromo(const string &s, int izq, int der) {
    if (izq >= der)      return true;
    if (s[izq] != s[der]) return false;
    return esPalindromo(s, izq + 1, der - 1);
}
```

Los dos índices como parámetros evitan copiar la cadena en cada llamada, que es lo que hace `substr` en el ejemplo anterior. Es el patrón habitual: **pasar el rango en vez de la subestructura**.

Torres de Hanói, el problema donde la recursión es claramente la solución natural:

```
void hanoi(int n, char origen, char destino, char auxiliar) {
    if (n == 0) return;
    hanoi(n - 1, origen, auxiliar, destino);
    cout << "Mover disco " << n << " de " << origen
         << " a " << destino << endl;
    hanoi(n - 1, auxiliar, destino, origen);
}
```

Mueve $2^n - 1$ discos. Escribirlo con bucles es posible y mucho más difícil de entender, que es el argumento a favor de la recursión.

6.5 Tipos de recursión

Tipo	Descripción	Ejemplo
Simple	una llamada por caso recursivo	factorial
Múltiple	varias llamadas	Fibonacci, Hanói
Final	la llamada recursiva es lo último que se hace	ver abajo
No final	queda trabajo después de la llamada	factorial
Mutua	dos funciones que se llaman entre sí	`par` e `impar`
Anidada	el argumento de la llamada es otra llamada	función de Ackermann

La **recursión final** —el resultado de la llamada es directamente el resultado de la función— tiene una propiedad útil: el compilador puede convertirla en un bucle y no consumir pila.

```
int factorialFinal(int n, int acumulado = 1) {
    if (n <= 1) return acumulado;
    return factorialFinal(n - 1, n * acumulado);   // nada despues
}
```

`g++ -O2` la convierte en un bucle. Es una optimización, no una garantía del lenguaje: sin optimizar, la pila crece igual.

6.6 Recursión frente a iteración

Toda función recursiva se puede escribir con bucles, y al revés. La elección es de claridad y de coste.

	Recursión	Iteración
Legibilidad	mayor si el problema es recursivo	mayor si no lo es
Memoria	un marco de pila por llamada	constante
Velocidad	algo menor, por el coste de llamar	mayor
Riesgo	desbordamiento de pila	bucle infinito

6.6.1 El caso donde la recursión ingenua es inaceptable

```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Es correcto y su coste es exponencial, $O(2^n)$, porque **recalcula los mismos valores una y otra vez**: `fibonacci(30)` invoca `fibonacci(5)` decenas de miles de veces. Con $n = 45$ ya tarda minutos.

La versión iterativa es lineal:

```
int fibonacciIterativo(int n) {
    if (n <= 1) return n;
    int anterior = 0, actual = 1;
    for (int i = 2; i <= n; i++) {
        int siguiente = anterior + actual;
        anterior = actual;
        actual = siguiente;
    }
    return actual;
}
```

La diferencia no es un porcentaje: para $n = 50$ son unas horas frente a unos microsegundos. Es el ejemplo que enseña que **elegir mal el algoritmo no se compensa con nada**.

La otra salida es guardar los resultados ya calculados —memorización— para no repetirlos, y eso convierte el árbol de llamadas en lineal sin abandonar la recursión. Es la puerta a la programación dinámica de cursos posteriores.

6.6.2 Cuándo usar cada una

Recursión cuando la definición del problema es recursiva y la iteración obligaría a gestionar una pila a mano: recorridos de árboles, divide y vencerás, vuelta atrás, Hanói.

Iteración cuando el problema es un recorrido lineal: sumar un vector, contar, buscar. Escribir `suma` recursiva es un ejercicio, no una buena solución.

6.7 Errores frecuentes

Error	Síntoma
Sin caso base	desbordamiento de pila
Caso base inalcanzable	desbordamiento de pila con ciertas entradas
No converger hacia el caso base	igual
Llamar con el mismo argumento	recursión infinita inmediata
Recalcular subproblemas	correcto, y exponencial
Copiar estructuras grandes en cada llamada	correcto, y muy lento

Los dos últimos son los peligrosos, porque el programa **funciona**: da el resultado correcto y tarda lo que no debe. Ninguna herramienta avisa de eso, y por eso el análisis del coste forma parte del diseño y no de la optimización posterior. El diseño de algoritmos recursivos y su análisis están desarrollados en Garrido Carrillo, 2005, Garrido Carrillo, 2016b y Savitch, 2017.

Capítulo 7

Temario práctico

Los seis bloques de prácticas del programa y los tres seminarios que los acompañan.

7.1 Seminario 1. El entorno de trabajo

```
g++ -Wall -Wextra -std=c++17 -g -o programa programa.cpp
./programa
```

Opción	Para qué
`-Wall -Wextra`	avisos; se tratan como errores
`-std=c++17`	fija la versión del lenguaje
`-g`	información para el depurador
`-o nombre`	nombre del ejecutable
`-O2`	optimiza; no se usa mientras se depura

Un programa que compila con avisos no está terminado. Los más frecuentes en esta asignatura, y lo que significan de verdad:

Aviso	Causa
`unused variable`	se declaró y no se usa: sobra, o falta código
`may be used uninitialized`	se lee antes de darle valor
`comparison between signed and unsigned`	comparar `int` con `size_t`
`control reaches end of non-void function`	falta un `return` en algún camino
`suggest parentheses around assignment`	se escribió `=` donde iba `==`

Redirección, que ahorra teclear la entrada en cada prueba:

```
./programa < entrada.txt > salida.txt
diff salida.txt esperada.txt
```

Si `diff` no imprime nada, la salida coincide. Es la forma más simple de prueba automática, y la que se usa en las entregas corregidas por juez.

7.2 Seminario 2. Test y depuración

7.2.1 Casos de prueba

Se eligen, no se improvisan:

Categoría	Ejemplo en un vector
Típico	cinco elementos desordenados
Vacío	`n = 0`
Un elemento	`n = 1`
Todos iguales	comprueba las comparaciones
Ya ordenado, y al revés	mejor y peor caso
Extremos del rango	el valor mínimo y el máximo del tipo

Los tres primeros encuentran la mayoría de los errores de esta asignatura.

7.2.2 assert

```
#include <cassert>

void probar() {
    assert(factorial(0) == 1);
    assert(factorial(5) == 120);
    assert(esPrimo(2));
    assert(!esPrimo(1));
}
```

`assert` aborta indicando fichero y línea si la condición es falsa, y desaparece al compilar con `-DNDEBUG`. Sirve para comprobar precondiciones e invariantes durante el desarrollo, no para validar la entrada del usuario: eso se comprueba con un `if` que siga funcionando en la versión final.

7.2.3 GDB

```
g++ -Wall -g -o prog prog.cpp
gdb ./prog
```

Orden	Efecto
`break main`	punto de ruptura
`run`	ejecuta
`next` / `step`	avanza sin entrar / entrando en funciones
`print x`	valor de una variable
`display x`	lo muestra tras cada paso
`backtrace`	pila de llamadas
`finish`	ejecuta hasta salir de la función actual

`backtrace` es la orden que resuelve los dos fallos más habituales del curso: tras un desbordamiento de pila muestra la recursión que no terminaba, y tras una violación de segmento indica desde dónde se llegó.

7.2.4 Valgrind

```
valgrind --leak-check=full ./prog
```

Detecta lecturas y escrituras fuera de un vector aunque el programa no falle. Es la única forma fiable de encontrar un desplazamiento en uno, porque `v[n]` no produce ningún error visible.

7.3 Seminario 3. Documentación de funciones

```
/**
 * @brief Cuenta cuantos elementos de un vector superan un umbral.
 *
 * @param v Vector de enteros. Precondicion: n >= 0.
 * @param n Numero de elementos utiles de v.
 * @param umbral Valor de comparacion.
 * @return El numero de elementos estrictamente mayores que umbral.
 */
int contarMayores(const int v[], int n, int umbral);
```

Se escribe **antes** del cuerpo, porque obliga a decidir qué hace la función antes de escribir cómo. Lo que documenta:

- `@brief`, en una frase y con un verbo.
- Cada parámetro, con su **precondición** si la tiene.
- Qué devuelve, incluido el valor para los casos especiales.

El comentario dice el **porqué** y el contrato; no repite lo que el código ya dice. `i++; // incrementa i` no aporta nada.

7.4 Bloque 1. Tipos de datos, expresiones y sentencias

Ejercicios sobre lo que el tema 1 explicaba, y las trampas que se comprueban ejecutando:

```
cout << 7 / 2;                          // 3
cout << 7 % 2;                          // 1
cout << 7.0 / 2;                        // 3.5
cout << static_cast<double>(7) / 2;     // 3.5
cout << 0.1 + 0.2 == 0.3;               // falso
```

Formato de salida:

```
#include <iomanip>
cout << fixed << setprecision(2) << 3.14159;   // 3.14
cout << setw(8) << 42;                          // alineado a la derecha
```

7.5 Bloque 2. Estructuras condicionales y repetitivas

Ejercicios característicos:

- Clasificar una nota en su calificación, con la escalera del tema 2 en el orden correcto.
- Contar dígitos de un número con `n /= 10` hasta que valga 0.
- Menú con `do-while` que repite hasta que se elige salir.
- Tabla de multiplicar con bucles anidados.
- Calcular el máximo común divisor con el algoritmo de Euclides.

```
int mcd(int a, int b) {
    while (b != 0) {
```

```
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

7.6 Bloque 3. Funciones

Reescribir los ejercicios del bloque 2 dividiéndolos en funciones, con su especificación. El ejercicio evalúa la **descomposición**, no el resultado: un programa correcto escrito entero dentro de `main` no cumple el objetivo.

Ejercicio de paso de parámetros que conviene tener claro:

```
void intercambiar(int &a, int &b) {   // por referencia: si funciona
    int t = a; a = b; b = t;
}

void noIntercambia(int a, int b) {    // por valor: no hace nada fuera
    int t = a; a = b; b = t;
}
```

7.7 Bloque 4. Vectores, matrices y registros

- Leer un vector, calcular su media, su máximo y su mínimo.
- Buscar de forma lineal y de forma dicotómica, y comparar el número de comparaciones en cada una.
- Implementar selección, inserción y burbuja, y contar comparaciones e intercambios de cada una sobre los mismos datos.
- Operaciones con matrices: suma, transpuesta, producto, comprobar si es simétrica.
- Vector de registros: gestionar una lista de alumnos, ordenarla por nota, buscar por nombre.

Contar operaciones en vez de medir tiempos es lo que hace comparable el resultado entre máquinas, y es lo que convierte la tabla de costes del tema 4 en algo comprobado.

7.8 Bloque 5. Clases

Convertir en clase lo que el bloque 4 hizo con vector y tamaño sueltos:

```
class ListaAlumnos {
private:
    static const int MAX = 100;
    Alumno datos_[MAX];
    int    n_;

public:
    ListaAlumnos() : n_(0) {}
    bool   anadir(const Alumno &a);
    bool   eliminar(int i);
    int    tamanio() const;
```

```
    Alumno elemento(int i) const;
    double media() const;
    void   ordenarPorNota();
};
```

La comprobación de que la clase está bien hecha: **cambiar la representación interna sin tocar el programa que la usa**. Si al sustituir el vector por otra estructura hay que modificar `main`, es que algo interno se había escapado por la interfaz.

7.9 Bloque 6. Recursividad

Los ejercicios habituales, de menos a más:

- Factorial, potencia y suma de dígitos.
- Invertir una cadena y comprobar si es palíndromo.
- Búsqueda dicotómica escrita de forma recursiva.
- Torres de Hanói.
- Fibonacci en las dos versiones, **midiendo el tiempo de las dos**.

El ejercicio de Fibonacci es el que cierra el curso, y su resultado es el que se recuerda: con $n = 45$, la versión recursiva ingenua tarda minutos y la iterativa es instantánea, escribiendo las dos lo mismo. Es la demostración de que el algoritmo pesa más que el lenguaje, la máquina o el compilador.

7.10 Cómo se entrega

```
g++ -Wall -Wextra -std=c++17 -o programa programa.cpp    # sin avisos
./programa < prueba.txt                                   # con los casos elegidos
valgrind ./programa < prueba.txt                          # sin accesos invalidos
```

Y lo que se revisa antes de entregar: nombres descriptivos, funciones con una responsabilidad, especificación en cada función, ninguna variable global no constante, ningún número mágico y ninguna línea repetida que pudiera ser una función. Los criterios están desarrollados en Martin, 2008 y McConnell, 2004; los ejercicios y su planteamiento, en Garrido Carrillo, 2005 y Gaddis et al., 2019.

Bibliografía

Gaddis, T., Walters, J., & Muganda, G. (2019). *Starting Out with C++: Early Objects* (10.ª ed.). Pearson.

Garrido Carrillo, A. (2005). *Fundamentos de Programación en C++*. Delta Publicaciones.

Garrido Carrillo, A. (2016a). *Fundamentos de programación con la STL*. Editorial Universidad de Granada.

Garrido Carrillo, A. (2016b). *Metodología de la Programación: de bits a objetos*. Editorial Universidad de Granada.

Lafore, R. (2005). *Object-Oriented Programming in C++* (4.ª ed.). Sams Publishing.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction* (2.ª ed.). Microsoft Press.

Savitch, W. (2017). *Resolución de problemas con C++*. Pearson.

Stroustrup, B. (2015). *The C++ Programming Language* (4.ª ed.). Addison-Wesley.