UNIVERSIDAD DE GRANADA

Metodología de la Programación

Temario de la asignatura

Ismael Sallami Moreno

Recursos Ingeniería Informática y Ade

Licencia

Metodología de la Programación

Ismael Sallami Moreno

https://elblogdeismael.github.io

Índice general

Capítulo 1

Punteros y memoria dinámica

Tema 1 del programa. El tipo puntero, su relación con los vectores y las cadenas, y la reserva de memoria en tiempo de ejecución.

1.1 El tipo de dato puntero

Un puntero guarda una **dirección de memoria**. Su tipo indica qué hay en esa dirección, y eso es lo que permite al compilador saber cuántos bytes leer y cómo interpretarlos.

```
int  x = 42;
int *p = &x;        // p apunta a x

cout << x;          // 42, el valor
cout << p;          // 0x7ffd..., la direccion
cout << *p;         // 42, el valor apuntado
*p = 100;
cout << x;          // 100: se modifico x a traves de p
```

Dos operadores, y conviene fijarlos desde el principio:

Operador	Nombre	Qué hace
`&`	dirección de	devuelve la dirección de un objeto
`*`	indirección	accede al objeto apuntado

El asterisco significa cosas distintas según dónde aparezca: en `int *p` forma parte de la declaración del tipo; en `*p = 100` es el operador de indirección. Es la fuente principal de confusión del tema.

1.1.1 `nullptr` y punteros colgantes

```
int *p = nullptr;       // no apunta a nada
if (p != nullptr) *p = 5;
```

Tres estados posibles de un puntero, y solo el primero es utilizable:

Estado	Qué contiene	Desreferenciarlo
Válido	la dirección de un objeto vivo	correcto
Nulo	`nullptr`	error detectable, suele abortar
Indeterminado o colgante	basura, o la dirección de algo que ya no existe	**comportamiento indefinido**

El tercer caso es el peligroso porque **puede funcionar**. Un puntero colgante apunta a memoria que el programa aún no ha reutilizado, así que el acceso da el valor antiguo durante las pruebas y otro distinto en la corrección.

```
int *malo() {
    int local = 5;
    return &local;      // local desaparece al volver
}
```

Esa función devuelve la dirección de una variable del marco de pila, que deja de existir. `-Wall` avisa de este caso concreto.

Se declara `nullptr` y no `NULL` ni `0`: `nullptr` tiene tipo propio y no se confunde con un entero al elegir entre funciones sobrecargadas.

1.1.2 Aritmética de punteros

Sumar 1 a un puntero avanza **un elemento**, no un byte:

```
int v[5] = {10, 20, 30, 40, 50};
int *p = v;

cout << *p;          // 10
cout << *(p + 1);    // 20
cout << *(p + 3);    // 40
```

El compilador multiplica por el tamaño del tipo, así que la aritmética funciona igual con `char` que con `double`. Restar dos punteros al mismo vector da el número de elementos entre ellos.

1.2 Vectores, matrices, cadenas y punteros

1.2.1 Un vector es un puntero a su primer elemento

El nombre de un vector se convierte en la dirección de `v[0]` en casi cualquier contexto. De ahí que estas dos expresiones sean equivalentes:

```
v[i]        // notacion de indice
*(v + i)    // aritmetica de punteros
```

Y de ahí también dos consecuencias que explican lo que en la asignatura anterior había que aceptar sin justificación:

- **Un vector se pasa a una función por referencia sin escribir `&`**: lo que se copia es la dirección, no los elementos.
- **`sizeof` de un parámetro vector no da el tamaño del vector**, sino el de un puntero, porque dentro de la función eso es lo que hay. Por eso el tamaño se pasa aparte.

```
void f(int v[]) {
    cout << sizeof(v);      // 8, el tamano de un puntero
}
```

1.2.2 Cadenas al estilo C

Un vector de `char` terminado en el carácter nulo `'\0'`:

```
char s[] = "Hola";     // 5 caracteres: 'H','o','l','a','\0'
cout << strlen(s);     // 4: no cuenta el terminador
```

El terminador ocupa espacio y no se cuenta en la longitud, y esa asimetría es la causa de la mayoría de los errores con cadenas C. Reservar `strlen(s)` bytes para copiar `s` deja fuera el `'\0'`, y la copia se lee más allá de su final.

`string` de C++ evita todo esto: gestiona su memoria, conoce su longitud y se copia y compara con los operadores. **Se usa `string` salvo que haya una razón concreta para no hacerlo**; las cadenas C se estudian porque están en muchas interfaces heredadas.

1.2.3 Matrices

Una matriz estática es un bloque contiguo recorrido por filas:

```
int m[3][4];
// m[i][j] esta en la posicion i*4 + j
```

Con memoria dinámica hay dos representaciones, y la diferencia importa:

```
// (a) vector de punteros a filas: las filas pueden estar dispersas
int **m = new int*[f];
for (int i = 0; i < f; i++) m[i] = new int[c];

// (b) un solo bloque: contiguo, se indexa a mano
int *m = new int[f * c];
// elemento (i,j): m[i * c + j]
```

La primera permite `m[i][j]` y hace dos accesos a memoria por elemento. La segunda es contigua, más rápida al recorrer y se libera con un solo `delete[]`.

1.3 Memoria dinámica

Las variables locales viven en la **pila** y su tamaño se fija al compilar. Cuando el tamaño depende de datos leídos en ejecución, hay que reservar en el **montículo**.

```
int n;
cin >> n;

int *v = new int[n];       // reserva
for (int i = 0; i < n; i++) v[i] = 0;
delete[] v;                // libera
v = nullptr;
```

Operación	Un objeto	Un vector
Reservar	`new int`	`new int[n]`
Liberar	`delete p`	`delete[] p`

Emparejar `new` con `delete` y `new[]` con `delete[]` no es una convención de estilo: mezclarlos es comportamiento indefinido. Liberar con `delete` lo reservado con `new[]` no llama a los destructores de los elementos y puede corromper el gestor de memoria.

Poner el puntero a `nullptr` después de liberar convierte un uso posterior en un fallo inmediato en vez de en un acceso a memoria liberada.

1.3.1 Pila y montículo

	Pila	Montículo
Quién gestiona	el compilador	el programador
Tamaño	fijado al compilar	decidido en ejecución
Vida	hasta que el bloque termina	hasta el `delete`
Velocidad	máxima	menor
Capacidad	megabytes	la memoria disponible
Error típico	desbordamiento por recursión	fugas y liberaciones dobles

1.3.2 Los cuatro errores

Error	Qué ocurre
Fuga de memoria	se reserva y no se libera; el consumo crece sin parar
Liberación doble	dos `delete` sobre el mismo puntero: corrompe el montículo
Uso tras liberar	acceso por un puntero ya liberado
Desbordamiento	escribir fuera del bloque reservado

Ninguno produce un error de compilación, y tres de los cuatro pueden no fallar en las pruebas. La herramienta que los encuentra es Valgrind:

```
g++ -Wall -g -o prog prog.cpp
valgrind --leak-check=full ./prog
```

Un programa correcto en esta asignatura termina con `All heap blocks were freed` y `ERROR SUMMARY: 0 errors`.

La fuga en un bucle es el caso que enseña por qué importa:

```
for (int i = 0; i < 1000000; i++) {
    int *p = new int[1000];
    // sin delete[]
}
```

Cada vuelta pierde 4 KB. Al millón de vueltas, 4 GB, y el programa muere sin haber hecho nada incorrecto a la vista.

1.4 Estructuras de datos simples

Con punteros y memoria dinámica se construyen estructuras cuyo tamaño no se conoce de antemano.

1.4.1 Vector dinámico

```
class VectorDinamico {
private:
    int *datos_;
    int  n_;          // elementos usados
    int  capacidad_;
```

```
    void redimensionar(int nuevaCapacidad) {
        int *nuevo = new int[nuevaCapacidad];
        for (int i = 0; i < n_; i++) nuevo[i] = datos_[i];
        delete[] datos_;
        datos_ = nuevo;
        capacidad_ = nuevaCapacidad;
    }

public:
    VectorDinamico() : datos_(new int[4]), n_(0), capacidad_(4) {}
    ~VectorDinamico() { delete[] datos_; }

    void anadir(int x) {
        if (n_ == capacidad_) redimensionar(capacidad_ * 2);
        datos_[n_++] = x;
    }
};
```

Duplicar la capacidad y no incrementarla en uno es lo que hace que añadir n elementos cueste $O(n)$ en total en vez de $O(n^2)$. Cada elemento se copia como mucho $\log n$ veces, y la suma de las copias es lineal.

Es, en esencia, cómo funciona `vector` de la biblioteca estándar.

1.4.2 Lista enlazada

```
struct Nodo {
    int   dato;
    Nodo *siguiente;
};

class Lista {
private:
    Nodo *cabeza_;

public:
    Lista() : cabeza_(nullptr) {}

    ~Lista() {
        while (cabeza_ != nullptr) {
            Nodo *aux = cabeza_;
            cabeza_ = cabeza_->siguiente;
            delete aux;
        }
    }

    void insertarInicio(int x) {
        Nodo *nuevo = new Nodo{x, cabeza_};
        cabeza_ = nuevo;
    }
```

```
};
```

El destructor guarda el siguiente **antes** de liberar el nodo actual. Al revés —liberar y luego leer `cabeza_->siguiente`— es un uso tras liberar, y suele funcionar durante las pruebas.

	Vector	Lista enlazada
Acceso por posición	$O(1)$	$O(n)$
Insertar al principio	$O(n)$	$O(1)$
Insertar al final	$O(1)$ amortizado	$O(1)$ con puntero al final
Memoria por elemento	el dato	el dato más un puntero
Localidad	contigua, aprovecha la caché	dispersa

La última fila es la que decide en la práctica: la lista gana en la teoría del coste y pierde en la máquina real, porque cada salto entre nodos es un posible fallo de caché.

1.4.3 Otras estructuras del tema

- **Pila**: se inserta y se extrae por el mismo extremo. Con lista, insertar y borrar al principio, las dos operaciones en $O(1)$.
- **Cola**: se inserta por un extremo y se extrae por el otro. Con lista, hace falta un puntero al último para que las dos sean $O(1)$.
- **Lista doblemente enlazada**: cada nodo apunta al anterior y al siguiente, lo que permite recorrer en los dos sentidos y borrar un nodo conocido en $O(1)$.

El desarrollo completo de punteros, memoria dinámica y estas estructuras está en Garrido Carrillo, 2016 y en Deitel y Deitel, 2017.

Capítulo 2

Funciones

Tema 2 del programa. La función `main`, el mecanismo de la pila que sostiene toda llamada, el paso de parámetros con punteros y referencias, y las funciones en línea y los punteros a función.

2.1 La función `main`

El punto de entrada del programa. Tiene dos formas admitidas:

```
int main();
int main(int argc, char *argv[]);
```

La segunda recibe los argumentos de la línea de órdenes:

Parámetro	Contenido
`argc`	número de argumentos, incluido el nombre del programa
`argv[0]`	el nombre con el que se invocó
`argv[1]...argv[argc-1]`	los argumentos
`argv[argc]`	`nullptr`

```
int main(int argc, char *argv[]) {
    if (argc != 3) {
        cerr << "Uso: " << argv[0] << " <entrada> <salida>" << endl;
        return 1;
    }
    string entrada = argv[1];
    string salida  = argv[2];
    // ...
    return 0;
}
```

Comprobar `argc` **antes** de leer `argv[1]` no es opcional: sin la comprobación, ejecutar el programa sin argumentos lee fuera del vector.

El valor devuelto es el **código de salida**: 0 significa éxito y cualquier otro valor, error. Lo consulta el intérprete de órdenes en `$?`, y es lo que permite encadenar programas en un guion.

2.2 La pila

Cada llamada a una función crea un **marco de activación** en la pila, que contiene:

Contenido	Para qué
Parámetros	los valores recibidos
Dirección de retorno	dónde continuar al volver
Variables locales	el estado de esta llamada
Registros salvados	los que la función va a usar

El marco se apila al llamar y se desapila al volver. De ahí salen los hechos que gobiernan todo lo demás:

- **Cada llamada tiene sus propias variables locales.** Es lo que hace funcionar la recursión.
- **Las locales dejan de existir al volver.** Devolver su dirección produce un puntero colgante, que es el error del tema 1.
- **La pila es finita.** Unos megabytes, así que una recursión demasiado profunda la agota. El programa aborta con desbordamiento de pila.
- **Un vector local grande no cabe.** `int v[1000000]` como variable local desborda; ese tamaño va al montículo.

```
direcciones altas
+--------------------------+
| marco de main            |
+--------------------------+
| marco de f, llamada por  |
| main                     |
+--------------------------+
| marco de g, llamada por f|  <- cima
+--------------------------+
direcciones bajas
```

En GDB, `backtrace` imprime exactamente esa pila, y es la orden que localiza de dónde vino una llamada que falló.

2.3 Paso de parámetros y devolución de resultados

Tres formas, y la elección tiene consecuencias distintas de las del curso anterior porque ahora entran los punteros.

2.3.1 Por valor

```
void f(int x);            // copia
void g(Objeto o);         // copia el objeto entero
```

Modificar el parámetro no afecta al argumento. Con objetos grandes la copia cuesta, y si el objeto gestiona memoria dinámica invoca su constructor de copia.

2.3.2 Por referencia

```
void incrementar(int &x)          { x++; }
void mostrar(const string &s)     { cout << s; }
```

Sin copia, y `const` impide modificar. **Es la opción por omisión para objetos**: no copia y el compilador comprueba la intención.

2.3.3 Por puntero

```
void incrementar(int *x) {
    if (x != nullptr) (*x)++;
}
```

Equivale a la referencia con dos diferencias que deciden cuándo usar cada una:

	Referencia	Puntero
Puede ser nula	no	sí
Se puede reasignar	no	sí
Sintaxis en la llamada	`f(x)`	`f(&x)`
Se ve en la llamada que puede modificar	no	sí

La regla: **referencia cuando el argumento siempre existe; puntero cuando puede no haberlo**. Un parámetro opcional se pasa por puntero precisamente porque `nullptr` es un valor válido que significa «no hay».

El paréntesis de `(*x)++` es obligatorio: `*x++` incrementa el puntero, no el valor, porque el postincremento tiene más prioridad que la indirección.

2.3.4 Devolver por referencia

```
int & elemento(int v[], int i) { return v[i]; }

elemento(v, 2) = 99;      // asigna a v[2]
```

Es lo que permite que una función aparezca a la izquierda de una asignación, y es como se implementa `operator[]` en el tema 3.

Nunca se devuelve una referencia a una variable local: deja de existir al volver. La referencia devuelta debe apuntar a algo que sobreviva a la llamada —un parámetro, un dato miembro o memoria dinámica—.

2.3.5 Devolver varios resultados

```
// (a) parametros de salida por referencia
void dividir(int a, int b, int &cociente, int &resto);

// (b) un registro
struct Division { int cociente, resto; };
Division dividir(int a, int b);
```

La segunda es preferible: los resultados van juntos, tienen nombre y no se pueden confundir de orden en la llamada.

2.4 Parámetros con valor por defecto

```
void imprimir(double x, int decimales = 2, bool alinear = false);

imprimir(3.14159);              // 2 decimales, sin alinear
imprimir(3.14159, 4);           // 4 decimales
imprimir(3.14159, 4, true);     // todo explicito
```

Dos reglas: **van al final** de la lista, porque los argumentos se emparejan por posición; y **el valor se escribe una sola vez**, en la declaración, no en la definición.

2.5 Funciones en línea

`inline` sugiere al compilador que sustituya la llamada por el cuerpo, ahorrando el coste de crear y destruir el marco:

```
inline int maximo(int a, int b) { return (a > b) ? a : b; }
```

Tres precisiones:

- **Es una sugerencia, no una orden.** El compilador decide, y con optimización activada pone en línea funciones que no llevan la palabra y descarta algunas que sí.
- **Solo compensa con funciones muy cortas.** Con una función grande, el código crece y empeora el aprovechamiento de la caché de instrucciones.
- **Su efecto real hoy es otro**: permite definir una función en una cabecera sin que el enlazador se queje de definiciones duplicadas. Los métodos definidos dentro de la declaración de una clase son implícitamente `inline`, y por eso se pueden escribir en el `.h`.

2.6 Punteros a función

Una función tiene dirección, y esa dirección se puede guardar:

```
int sumar(int a, int b) { return a + b; }
int restar(int a, int b) { return a - b; }

int (*operacion)(int, int) = sumar;
cout << operacion(3, 4);      // 7
operacion = restar;
cout << operacion(3, 4);      // -1
```

Los paréntesis de `(*operacion)` son necesarios: sin ellos, `int *operacion(int, int)` declara una función que devuelve un puntero a entero.

Para qué sirven:

Pasar comportamiento a una función. Una ordenación que recibe el criterio de comparación sirve para cualquier orden sin duplicar el algoritmo:

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

bool creciente(int a, int b)   { return a < b; }
bool decreciente(int a, int b) { return a > b; }

ordenar(v, n, creciente);
ordenar(v, n, decreciente);
```

Es exactamente lo que hacen `qsort` de C y `sort` de C++, y es el primer ejemplo de un algoritmo genérico: el recorrido y los intercambios se escriben una vez y el criterio se pasa desde fuera.

Tablas de funciones. Un menú puede ser un vector de punteros a función indexado por la opción, en lugar de un `switch` con una llamada por caso.

Retrollamadas. Una biblioteca recibe una función del usuario y la invoca cuando ocurre algo.

2.6.1 Alternativas modernas

`typedef` o `using` hacen legible el tipo:

```
using Comparador = bool (*)(int, int);
void ordenar(int v[], int n, Comparador antes);
```

Y las **expresiones lambda** permiten escribir la función en el punto de uso:

```
sort(v, v + n, [](int a, int b) { return a > b; });
```

Un puntero a función solo puede apuntar a una función libre y no lleva estado asociado; una lambda sí puede capturar variables del entorno. Por eso las bibliotecas modernas reciben objetos función o lambdas, y no punteros a función. El tratamiento de estos mecanismos está en Garrido Carrillo, 2016 y en Stroustrup, 2013.

Capítulo 3

Tipos de datos abstractos en C++: clases

Tema 3 del programa. Diseñar un tipo nuevo separando lo que promete de cómo está hecho, y qué hay que escribir cuando ese tipo gestiona memoria dinámica.

3.1 Abstracción y diseño de clases

Un **tipo de dato abstracto** es un tipo definido por sus operaciones y no por su representación. Quien lo usa conoce qué hace cada operación; cómo está hecho por dentro es privado y puede cambiar.

La separación tiene dos caras:

Cara	Qué contiene	Quién la ve
Interfaz	los métodos públicos y lo que prometen	quien usa la clase
Implementación	los datos y el cuerpo de los métodos	solo la clase

```
class Conjunto {
private:
    int *datos_;
    int  n_;
    int  capacidad_;

    int posicion(int x) const;         // detalle interno

public:
    Conjunto();
    Conjunto(const Conjunto &otro);
    ~Conjunto();
    Conjunto & operator=(const Conjunto &otro);

    bool contiene(int x) const;
    bool insertar(int x);
    bool borrar(int x);
    int  cardinal() const;
};
```

Que `posicion` sea privado es deliberado: devuelve un índice del vector interno, y exponerlo ataría el código de fuera a que la representación sea un vector.

3.1.1 Atributos y métodos

Los **atributos** guardan el estado y son privados. Los **métodos** son las operaciones, y solo se hacen públicos los que el problema necesita.

Escribir mecánicamente un consultor y un modificador por cada atributo anula la encapsulación: si todo se lee y se escribe desde fuera, la clase es un registro con más líneas. La pregunta no es «qué datos tengo» sino «qué operaciones ofrezco».

3.1.2 El invariante

Un **invariante de clase** es una propiedad que se cumple en todo objeto válido entre operaciones. En el `Conjunto` de arriba:

`0 <= n_ <= capacidad_,  datos_ != nullptr,  los n_ primeros no se repiten`

El constructor lo establece y cada método lo mantiene. Escribirlo como comentario en la clase es lo que convierte la corrección de los métodos en algo comprobable por partes: cada uno puede suponer el invariante al entrar y debe dejarlo cierto al salir.

3.1.3 `const` en los métodos

Un método que no modifica el objeto se marca `const`, y **solo esos se pueden llamar sobre un objeto constante**:

```
int  cardinal() const;      // consulta
bool insertar(int x);       // modifica
```

Olvidar el `const` en un consultor rompe todo el código que recibe el objeto por referencia constante, que es la forma habitual de pasarlo. Es un error de compilación, y por eso conviene: el compilador comprueba lo que si no habría que recordar.

3.2 Clases que gestionan memoria dinámica

Aquí está el contenido propio del tema, y lo que lo separa de las clases del curso anterior.

3.2.1 El problema

Si un atributo es un puntero a memoria reservada, el constructor de copia y el operador de asignación que el compilador genera **copian el puntero, no lo apuntado**. Se llama copia superficial, y produce dos objetos que comparten el mismo bloque:

```
Conjunto a;
a.insertar(5);
Conjunto b = a;      // copia superficial: b.datos_ == a.datos_
```

Consecuencias, las tres graves:

1. Modificar `b` modifica `a`.
2. Al destruirse los dos, el bloque se libera dos veces y el montículo se corrompe.
3. Si uno redimensiona, el otro queda con un puntero colgante.

3.2.2 La regla de los tres

Una clase que gestiona un recurso necesita **destructor, constructor de copia y operador de asignación**. Si hace falta uno, hacen falta los tres.

Constructor y destructor

```
Conjunto::Conjunto()
    : datos_(new int[4]), n_(0), capacidad_(4) {}

Conjunto::~Conjunto() {
    delete[] datos_;
}
```

El destructor se llama solo: al salir del ámbito para un objeto local, al hacer `delete` para uno dinámico, y al destruirse el objeto que lo contiene. Es lo que garantiza que la memoria se libere aunque la función termine por un camino inesperado.

Constructor de copia

```
Conjunto::Conjunto(const Conjunto &otro)
    : datos_(new int[otro.capacidad_]),
      n_(otro.n_),
      capacidad_(otro.capacidad_) {
    for (int i = 0; i < n_; i++) datos_[i] = otro.datos_[i];
}
```

Reserva memoria propia y copia el contenido: copia **profunda**. Se invoca al inicializar un objeto con otro, al pasar por valor y al devolver por valor.

Operador de asignación

```
Conjunto & Conjunto::operator=(const Conjunto &otro) {
    if (this != &otro) {                // 1. autoasignacion
        delete[] datos_;                // 2. liberar lo propio
        capacidad_ = otro.capacidad_;   // 3. reservar y copiar
        n_ = otro.n_;
        datos_ = new int[capacidad_];
        for (int i = 0; i < n_; i++) datos_[i] = otro.datos_[i];
    }
    return *this;                       // 4. permitir encadenar
}
```

Los cuatro pasos son obligatorios y cada uno arregla un fallo concreto:

Paso	Si falta
Comprobar la autoasignación	`a = a` libera los datos y luego los copia de sí mismo, ya liberados
Liberar lo propio	fuga de memoria: el bloque anterior se pierde

Paso	Si falta
Copia profunda	los dos objetos comparten el bloque
Devolver `*this`	`a = b = c` no compila

La diferencia con el constructor de copia: el constructor trabaja sobre un objeto que aún no existe, así que no tiene nada que liberar; el operador trabaja sobre uno ya construido, y por eso empieza liberando.

3.2.3 Cómo se comprueba

```
void probarCopia() {
    Conjunto a;
    a.insertar(1); a.insertar(2);

    Conjunto b = a;          // constructor de copia
    b.insertar(3);
    assert(a.cardinal() == 2);   // a no debe haber cambiado
    assert(b.cardinal() == 3);

    Conjunto c;
    c = a;                   // asignacion
    c = c;                   // autoasignacion: no debe romper nada
    assert(c.cardinal() == 2);
}
```

Y después, Valgrind. Una clase correcta de este tema termina con cero fugas y cero errores.

3.3 Sobrecarga de operadores

Dar significado a los operadores del lenguaje para el tipo nuevo. Es lo que hace que un tipo definido por el programador se use como uno básico.

3.3.1 Métodos frente a funciones externas

```
class Conjunto {
public:
    bool operator==(const Conjunto &otro) const;
    Conjunto operator+(const Conjunto &otro) const;   // union
    int  operator[](int i) const;
};
```

Se declara **como método** cuando el operando izquierdo es el objeto, y **como función externa** cuando no lo es:

```
ostream & operator<<(ostream &os, const Conjunto &c) {
    os << "{";
    for (int i = 0; i < c.cardinal(); i++) {
        if (i > 0) os << ", ";
        os << c[i];
    }
```

```
    return os << "}";
}
```

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

3.3.2 `operator[]` en dos versiones

```
int & operator[](int i);              // para objetos no constantes
int   operator[](int i) const;        // para objetos constantes
```

La primera devuelve una referencia, así que `v[0] = 5` funciona. La segunda devuelve una copia y se aplica a objetos `const`. Las dos hacen falta, y es el mismo motivo por el que el tema 2 hablaba de devolver por referencia.

3.3.3 Reglas

Regla	Motivo
El operador debe significar lo que significa	`+` que resta compila y hace ilegible el programa
No todos se pueden sobrecargar	`.`, `::`, `?:` y `sizeof` no
No se puede inventar un operador nuevo	solo redefinir los existentes
Al menos un operando debe ser del tipo propio	no se puede cambiar `int + int`
Si se define `==`, definir también `!=`	o el uso se vuelve asimétrico

3.4 Amistad

`friend` da a una función externa acceso a lo privado:

```
class Conjunto {
    friend ostream & operator<<(ostream &, const Conjunto &);
};
```

Rompe la encapsulación a propósito, así que se usa lo mínimo. En el caso del operador de salida a menudo ni hace falta: si la clase ya ofrece los consultores necesarios, la función se escribe sin ser amiga, que es lo que hace el ejemplo de más arriba.

3.5 Compilación separada

Una clase se reparte en dos ficheros:

```
// conjunto.h
#ifndef CONJUNTO_H
#define CONJUNTO_H

class Conjunto {
    // declaraciones
};

#endif
```

```
// conjunto.cpp
#include "conjunto.h"
```

```
Conjunto::Conjunto() : datos_(new int[4]), n_(0), capacidad_(4) {}
// resto de definiciones
```

El operador `::` indica a qué clase pertenece cada definición. Y en la cabecera no se escribe `using namespace std`, porque se lo impone a todo el que la incluya. El diseño de tipos abstractos y la gestión de recursos en clases están desarrollados en Garrido Carrillo, 2016 y en Deitel y Deitel, 2017; el detalle del lenguaje, en Stroustrup, 2013.

Capítulo 4

Gestión de entrada/salida. Ficheros

Tema 4 del programa. Los flujos de C++, las operaciones comunes a todos ellos, y los tres destinos que pueden tener: la consola, una cadena y un fichero.

4.1 Flujos de entrada/salida

Un **flujo** es una secuencia de bytes con un origen o un destino. C++ los organiza en una jerarquía de clases, y esa es la razón de que la misma sintaxis sirva para la consola, para un fichero y para una cadena.

Clase	Definida en	Para qué
`istream`	`<iostream>`	entrada genérica
`ostream`	`<iostream>`	salida genérica
`ifstream`	`<fstream>`	lectura de fichero
`ofstream`	`<fstream>`	escritura en fichero
`fstream`	`<fstream>`	lectura y escritura
`istringstream`	`<sstream>`	lectura desde una cadena
`ostringstream`	`<sstream>`	escritura sobre una cadena

Los objetos predefinidos:

Objeto	Qué es	Almacenamiento intermedio
`cin`	entrada estándar	sí
`cout`	salida estándar	sí
`cerr`	salida de error	no
`clog`	registro	sí

`cerr` no usa búfer, así que sus mensajes salen inmediatamente aunque el programa aborte después. Es la razón por la que los errores van ahí y no a `cout`: la salida de `cout` puede perderse en el búfer si el programa termina de forma anómala.

Y hay otra razón, más práctica: `cout` y `cerr` son flujos distintos, así que al redirigir la salida a un fichero los errores siguen apareciendo en la pantalla.

4.1.1 La consecuencia de la jerarquía

Una función que recibe `ostream &` funciona con la consola, con un fichero y con una cadena sin cambiar una línea:

```
void informe(ostream &os, const Datos &d) {
    os << "Total: " << d.total() << endl;
}

informe(cout, d);                // a la consola
ofstream f("salida.txt");
informe(f, d);                   // a un fichero
ostringstream s;
informe(s, d);                   // a una cadena
```

Es el polimorfismo de la biblioteca aplicado a algo cotidiano, y es la razón por la que **una función que escribe nunca debe usar `cout` directamente**: recibe el flujo y así se puede probar redirigiéndola a una cadena.

4.2 Operaciones básicas con flujos

4.2.1 Operadores de inserción y extracción

```
os << valor;      // insercion: escribe
is >> variable;   // extraccion: lee
```

Devuelven el flujo, lo que permite encadenar, y **se convierten a `bool` según el estado**, que es lo que hace funcionar el idioma de lectura de más abajo.

`>>` salta los espacios en blanco iniciales y se detiene en el primer separador, así que no lee una línea con espacios.

4.2.2 Lectura de líneas

```
string linea;
while (getline(is, linea)) {
    // procesar linea
}
```

Ese bucle es el patrón estándar de lectura, y funciona porque `getline` devuelve el flujo y el flujo se evalúa como falso cuando falla.

El problema clásico de mezclar los dos operadores:

```
int n;
cin >> n;              // deja el salto de linea en el flujo
string nombre;
getline(cin, nombre);  // lee una cadena vacia
```

Se corrige descartando lo que queda de la línea:

```
cin >> n;
cin.ignore(numeric_limits<streamsize>::max(), '\n');
getline(cin, nombre);
```

4.2.3 Estado del flujo

Bandera	Consulta	Significado
`good`	`is.good()`	todo correcto
`eof`	`is.eof()`	se alcanzó el final
`fail`	`is.fail()`	la última operación falló, formato incorrecto
`bad`	`is.bad()`	error irrecuperable

Una vez que un flujo entra en estado de error, **las operaciones siguientes no hacen nada**. Se limpia con `is.clear()`, y suele hacer falta descartar además lo que quedaba en el flujo:

```
int n;
while (!(cin >> n)) {
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cerr << "Numero no valido, repita: ";
}
```

4.2.4 El error de leer con `eof`

```
while (!f.eof()) {          // incorrecto
    f >> x;
    procesar(x);
}
```

`eof()` se activa **después** de intentar leer más allá del final, no antes. Con ese bucle, la última lectura falla, `x` conserva el valor anterior y se procesa **dos veces el último elemento**. Es el error más frecuente del tema, y no falla: da un resultado incorrecto.

La forma correcta usa el resultado de la propia lectura como condición:

```
while (f >> x) {
    procesar(x);
}
```

4.2.5 Formato

```
#include <iomanip>

cout << fixed << setprecision(2) << 3.14159;   // 3.14
cout << setw(10) << left  << "Nombre";
cout << setw(8)  << right << 42;
cout << setfill('0') << setw(3) << 7;          // 007
```

`setw` afecta **solo a la siguiente inserción**; los demás manipuladores son persistentes hasta que se cambien. Es la asimetría que produce tablas desalineadas.

4.3 Flujos asociados a cadenas

Un `stringstream` trata una cadena como un flujo, y resuelve dos problemas habituales.

4.3.1 Convertir y trocear

```
#include <sstream>
```

```
// De cadena a numero
string s = "42";
int n;
istringstream(s) >> n;

// Trocear una linea por espacios
istringstream iss("uno dos tres");
string palabra;
while (iss >> palabra) {
    cout << palabra << endl;
}

// Trocear por un separador concreto
istringstream campos("Ana;19;8.5");
string nombre, edad, nota;
getline(campos, nombre, ';');
getline(campos, edad,   ';');
getline(campos, nota,   ';');
```

El tercer parámetro de `getline` es el delimitador, y es lo que permite leer un fichero de campos separados sin escribir un analizador a mano.

4.3.2 Construir una cadena

```
ostringstream oss;
oss << "Alumno " << nombre << " con nota " << fixed << setprecision(2) << nota;
string mensaje = oss.str();
```

Da acceso a todo el formato de los flujos para producir una cadena, que es lo que las funciones de conversión sueltas no ofrecen.

4.4 Flujos asociados a ficheros

4.4.1 Abrir y cerrar

```
#include <fstream>

ifstream entrada("datos.txt");
if (!entrada) {
    cerr << "No se pudo abrir datos.txt" << endl;
    return 1;
}
// ...
entrada.close();
```

Comprobar que se abrió es obligatorio: un fichero que no existe, o sin permisos, deja el flujo en estado de error y las lecturas devuelven basura sin avisar de nada.

El destructor cierra el fichero, así que `close` explícito solo hace falta para cerrarlo antes de que el objeto salga del ámbito. Que el destructor libere el recurso es el mismo principio que el destructor del tema 3.

4.4.2 Modos de apertura

Modo	Efecto
`ios::in`	lectura
`ios::out`	escritura; **trunca** el fichero
`ios::app`	escritura al final, sin truncar
`ios::binary`	modo binario
`ios::ate`	se posiciona al final tras abrir

```
ofstream f("registro.log", ios::app);   // anade, no borra
```

Abrir para escritura sin `ios::app` **borra el contenido**. Es el error que hace perder un fichero de datos por escribir el modo equivocado.

4.4.3 Ficheros de texto

```
// Escribir
ofstream salida("alumnos.txt");
for (int i = 0; i < n; i++) {
    salida << v[i].nombre << ";" << v[i].edad << ";" << v[i].nota << endl;
}
salida.close();

// Leer
ifstream entrada("alumnos.txt");
string linea;
while (getline(entrada, linea)) {
    istringstream campos(linea);
    string nombre, edad, nota;
    getline(campos, nombre, ';');
    getline(campos, edad,   ';');
    getline(campos, nota,   ';');
    // convertir y guardar
}
```

Leer línea a línea y trocear con `istringstream` es más robusto que leer campo a campo del fichero: una línea mal formada no descoloca el resto del proceso.

4.4.4 Ficheros binarios

Se escribe la representación interna, sin convertir a texto:

```
struct Registro { int id; double valor; };

// Escribir
ofstream f("datos.bin", ios::binary);
Registro r = {1, 3.5};
f.write(reinterpret_cast<const char *>(&r), sizeof(r));

// Leer
ifstream g("datos.bin", ios::binary);
```

```
Registro leido;
g.read(reinterpret_cast<char *>(&leido), sizeof(leido));
```

	Texto	Binario
Legible	sí	no
Tamaño	mayor	menor
Velocidad	menor, hay conversión	mayor
Acceso directo	difícil, líneas de longitud variable	inmediato si los registros son de tamaño fijo
Portable entre máquinas	sí	**no**

La última fila es la limitación seria: un fichero binario depende del tamaño de los tipos, del relleno de la estructura y del orden de los bytes. Un fichero escrito en una máquina puede no leerse en otra. Y **no se puede escribir así una clase con punteros**: lo que se guardaría son direcciones, que no significan nada en la siguiente ejecución.

4.4.5 Acceso directo

Con registros de tamaño fijo se salta a cualquiera sin leer los anteriores:

```
f.seekg(i * sizeof(Registro), ios::beg);   // posicionar para leer
f.read(reinterpret_cast<char *>(&r), sizeof(r));
```

Función	Para qué
`seekg`, `tellg`	posición de lectura
`seekp`, `tellp`	posición de escritura
`ios::beg`, `ios::cur`, `ios::end`	origen del desplazamiento

Es lo que permite modificar el registro i de un fichero de un millón sin tocar los demás, y es la razón práctica de usar formato binario. La gestión de flujos y ficheros está desarrollada en Garrido Carrillo, 2016, Deitel y Deitel, 2017 y Gaddis et al., 2019.

Capítulo 5

Seminarios

Los cinco seminarios del programa. Cubren las herramientas con las que se construye, se depura y se documenta un proyecto, que es la parte de la asignatura que no es lenguaje sino método.

5.1 Seminario 1. El modelo de compilación en C++

5.1.1 Las cuatro fases

Fase	Programa	Entrada	Salida	Cómo detenerse
Preprocesado	`cpp`	`.cpp`	texto expandido	`g++ -E`
Compilación	`cc1plus`	texto expandido	ensamblador `.s`	`g++ -S`
Ensamblado	`as`	`.s`	objeto `.o`	`g++ -c`
Enlazado	`ld`	`.o` y bibliotecas	ejecutable	`g++`

El **preprocesador** no entiende C++: manipula texto. Sustituye `#include` por el contenido del fichero, expande las macros y resuelve la compilación condicional. De ahí que un error en una macro señale una línea que no es la culpable.

El **enlazador** resuelve los símbolos: cada `.o` deja anotado qué usa y no define, y el enlazador busca esas definiciones en los demás objetos y en las bibliotecas.

5.1.2 Compilación separada

```
g++ -Wall -c principal.cpp     # -> principal.o
g++ -Wall -c conjunto.cpp      # -> conjunto.o
g++ principal.o conjunto.o -o programa
```

Lo que gana: al cambiar un `.cpp` solo se recompila ese, y en un proyecto grande eso es la diferencia entre segundos y minutos.

El reparto entre cabecera y fuente:

Fichero	Contiene
`.h`	declaraciones: clases, prototipos, constantes, plantillas
`.cpp`	definiciones: cuerpos de las funciones y métodos

5.1.3 Guardián de inclusión

```
#ifndef CONJUNTO_H
#define CONJUNTO_H
// ...
#endif
```

Sin él, una cabecera incluida dos veces —directamente y a través de otra— duplica las declaraciones y el compilador falla. `#pragma once` hace lo mismo en una línea y no es estándar, aunque todos los compiladores lo admiten.

5.1.4 Errores de compilación y de enlazado

Distinguirlos ahorra tiempo, porque se arreglan en sitios distintos:

Mensaje	Fase	Causa habitual
`'X' was not declared in this scope`	compilación	falta el `#include` o hay una errata
`undefined reference to 'X'`	**enlazado**	el símbolo se declaró y no se definió, o falta el `.o`
`multiple definition of 'X'`	enlazado	una definición en una cabecera incluida dos veces
`redefinition of class X`	compilación	falta el guardián de inclusión

`undefined reference to 'Conjunto::insertar(int)'` significa casi siempre que se declaró el método y no se escribió su cuerpo, o que se olvidó `Conjunto::` delante de la definición.

5.2 Seminario 2. Gestión automatizada de proyectos

5.2.1 `make`

Un fichero `makefile` describe qué depende de qué y cómo se construye. `make` recompila **solo lo que ha cambiado**, comparando fechas.

```
CXX      = g++
CXXFLAGS = -Wall -Wextra -std=c++17 -g
OBJ      = principal.o conjunto.o

programa: $(OBJ)
    $(CXX) $(OBJ) -o programa

principal.o: principal.cpp conjunto.h
    $(CXX) $(CXXFLAGS) -c principal.cpp

conjunto.o: conjunto.cpp conjunto.h
    $(CXX) $(CXXFLAGS) -c conjunto.cpp

clean:
```

```
    rm -f $(OBJ) programa

.PHONY: clean
```

Tres cosas que hay que acertar:

- **Las órdenes se indentan con un tabulador**, no con espacios. Con espacios `make` da un error que no explica lo que pasa.
- **Las cabeceras van en las dependencias.** Si `principal.o` no depende de `conjunto.h`, cambiar la cabecera no recompila nada y el programa se enlaza con código que ya no corresponde. Es el error más difícil de diagnosticar del seminario, porque el binario resultante es incoherente.
- **`.PHONY`** marca los objetivos que no producen un fichero con ese nombre. Sin él, un fichero llamado `clean` impediría ejecutar la regla.

Estructura habitual de un proyecto de la asignatura:

```
proyecto/
  include/     cabeceras
  src/         fuentes
  obj/         objetos
  bin/         ejecutable
  doc/         documentacion generada
  makefile
```

5.3 Seminario 3. Modularización y bibliotecas

Un **módulo** agrupa lo relacionado: una cabecera con la interfaz y un fuente con la implementación. Un módulo bien hecho tiene alta cohesión —todo lo suyo sirve al mismo propósito— y bajo acoplamiento —depende de pocos módulos—.

5.3.1 Bibliotecas estáticas y el programa `ar`

```
g++ -Wall -c conjunto.cpp lista.cpp
ar rcs libestructuras.a conjunto.o lista.o
g++ principal.cpp -L. -lestructuras -o programa
```

Opción de `ar`	Efecto
`r`	inserta o reemplaza
`c`	crea el archivo sin avisar
`s`	escribe el índice de símbolos

Y en el enlazado: `-L.` añade el directorio actual a la búsqueda, y `-lestructuras` busca `libestructuras.a`. La convención del nombre —`lib` delante y `.a` detrás— es obligatoria para que `-l` lo encuentre.

	Estática `.a`	Dinámica `.so`
Cuándo se enlaza	al construir	al ejecutar
Tamaño del ejecutable	mayor	menor
Actualizar la biblioteca	hay que reenlazar	basta con sustituir el fichero
Dependencias en ejecución	ninguna	la biblioteca debe estar

Otras herramientas del seminario: `nm` lista los símbolos de un objeto o biblioteca, y `ar t` su contenido. `nm` es lo que resuelve un `undefined reference` cuando no está claro si el símbolo existe o se llama de otra forma.

5.4 Seminario 4. Gestión de errores y depuración

Tres mecanismos, para tres clases de problema distintas.

5.4.1 Devolución de valores de error

```
bool leerConfiguracion(const string &fichero, Config &c);
```

Simple y explícito. Su defecto es que **el valor se puede ignorar**, y entonces el programa sigue con datos inválidos.

5.4.2 Aserciones

```
#include <cassert>

int elemento(int i) const {
    assert(i >= 0 && i < n_);
    return datos_[i];
}
```

Comprueban lo que **nunca debería fallar**: precondiciones e invariantes. Si falla, el error está en el programa, no en los datos.

Desaparecen al compilar con `-DNDEBUG`, y de ahí la regla que las gobierna: **nunca se pone un efecto necesario dentro de un `assert`**.

```
assert(insertar(x));      // en la version final NO se inserta nada
```

Y no sirven para validar la entrada del usuario: eso se comprueba con un `if` que siga estando en la versión final.

5.4.3 Excepciones

Para errores que el punto donde ocurren no puede resolver:

```
#include <stdexcept>

int Conjunto::elemento(int i) const {
    if (i < 0 || i >= n_) {
        throw out_of_range("Conjunto::elemento: indice fuera de rango");
    }
    return datos_[i];
}

// En quien llama
try {
    cout << c.elemento(50);
} catch (const out_of_range &e) {
    cerr << "Error: " << e.what() << endl;
}
```

Excepción estándar	Cuándo
`out_of_range`	índice fuera de rango
`invalid_argument`	argumento con valor no admitido
`runtime_error`	fallo detectado en ejecución
`bad_alloc`	`new` no pudo reservar

Se capturan **por referencia constante**: por valor se copia y se pierde el tipo derivado.

Ventaja: el error no se puede ignorar por descuido, y la información viaja desde donde ocurre hasta donde se sabe qué hacer. Inconveniente: el flujo deja de ser lineal, y **una excepción que sale de una función salta el código que quedaba**, incluidos los `delete`. Esa es la razón profunda de que los recursos se liberen en un destructor y no a mano.

5.4.4 Cuál usar

Situación	Mecanismo
Fallo esperable y frecuente	valor de retorno
Error del programador	aserción
Error excepcional que aquí no se puede resolver	excepción
Entrada del usuario incorrecta	comprobación con `if`

5.4.5 Depuración

```
g++ -Wall -Wextra -g -o prog prog.cpp
gdb ./prog
valgrind --leak-check=full ./prog
```

Orden de GDB	Efecto
`break conjunto.cpp:42`	punto de ruptura
`run`	ejecuta
`next` / `step`	avanza sin entrar / entrando
`print *p`	valor apuntado
`backtrace`	pila de llamadas
`watch n_`	detiene cuando cambia un dato
`finish`	ejecuta hasta salir de la función

`watch` es la orden que encuentra el atributo que alguien modifica sin permiso, y `backtrace` la que dice desde dónde se llegó a la violación de segmento.

Salida de Valgrind sobre un programa correcto de esta asignatura:

```
All heap blocks were freed -- no leaks are possible
ERROR SUMMARY: 0 errors from 0 contexts
```

Cualquier otra cosa es un fallo, aunque el programa dé el resultado esperado.

5.5 Seminario 5. Documentación de software

5.5.1 Doxygen

```
/**
 * @file conjunto.h
 * @brief Conjunto de enteros sin repeticiones.
 * @author Ismael Sallami Moreno
 */

/**
 * @brief Inserta un elemento en el conjunto.
 *
 * Si el elemento ya estaba, el conjunto no cambia.
 *
 * @param x Elemento a insertar.
 * @return true si se inserto, false si ya estaba.
 * @pre El conjunto esta correctamente construido.
 * @post cardinal() >= el valor anterior.
 */
bool insertar(int x);
```

```
doxygen -g            # genera Doxyfile
doxygen Doxyfile      # genera la documentacion
```

Etiqueta	Para qué
`@brief`	una frase
`@param`	cada parámetro
`@return`	qué devuelve
`@pre`, `@post`	precondición y postcondición
`@throw`	qué excepciones lanza
`@file`, `@author`	cabecera del fichero

5.5.2 Qué documentar

- **El contrato**: qué hace, qué recibe, qué devuelve, qué exige y qué garantiza.
- **El porqué** de una decisión que no se deduce del código.
- **Las excepciones** que puede lanzar.

Y qué no: repetir lo que el código dice. `i++; // incrementa i` no aporta nada, y un comentario que se desactualiza es peor que ninguno, porque miente con autoridad.

La documentación se escribe **antes** que el cuerpo, porque obliga a decidir el contrato antes de la implementación. Es la misma disciplina que la especificación de la asignatura anterior, ahora con herramientas que la extraen. Estos seminarios están desarrollados en Garrido Carrillo, 2016 y en Garrido Carrillo, 2017.

Capítulo 6

Temario práctico

Las prácticas sobre los cuatro temas y el proyecto informático de programación, que es lo que la guía docente sitúa como cierre de la asignatura.

6.1 Prácticas sobre el tema 1. Punteros y memoria dinámica

Ejercicios característicos:

- Recorrer un vector con aritmética de punteros en vez de con índices, y comprobar que `v[i]` y `*(v+i)` producen el mismo código.
- Reservar un vector cuyo tamaño se lee del teclado, usarlo y liberarlo.
- Implementar `strlen`, `strcpy` y `strcmp` sobre cadenas al estilo C, con el terminador nulo gestionado a mano.
- Matriz dinámica en las dos representaciones —vector de punteros y bloque contiguo— y comparar el tiempo de recorrerlas.

El ejercicio que enseña lo que hay que retener:

```
int *v = new int[n];
// ...
delete[] v;
v = nullptr;      // usarlo despues falla en el acto, no en silencio
```

Y el que enseña lo contrario:

```
int *v = new int[n];
v = new int[m];   // el primer bloque se ha perdido: fuga
```

Reasignar un puntero sin liberar antes pierde el bloque anterior. Valgrind lo señala con `definitely lost`.

Toda práctica de este tema se entrega con Valgrind limpio. Un programa que da el resultado correcto y pierde memoria no está terminado.

6.2 Prácticas sobre el tema 2. Funciones

- Un programa que lea sus parámetros de la línea de órdenes y compruebe `argc` antes de usarlos.
- Comparar el paso por valor, por referencia y por puntero sobre la misma función, imprimiendo las direcciones para ver qué se copia.
- Escribir una ordenación que reciba el criterio como puntero a función y usarla con varios criterios sin tocar el algoritmo.
- Una tabla de opciones de menú resuelta con un vector de punteros a función en lugar de un

switch.

```
struct Opcion { const char *texto; void (*accion)(); };

Opcion menu[] = {
    {"Alta",   alta},
    {"Baja",   baja},
    {"Listar", listar}
};

// Ejecutar la opcion i
menu[i].accion();
```

Añadir una opción es una línea del vector y una función. Con `switch` habría que tocar dos sitios.

6.3 Prácticas sobre el tema 3. Clases

La práctica central de la asignatura: una clase que gestiona memoria dinámica, con la regla de los tres completa.

```
class Cadena {
private:
    char *datos_;
    int   longitud_;

public:
    Cadena();
    Cadena(const char *s);
    Cadena(const Cadena &otra);              // constructor de copia
    ~Cadena();                               // destructor
    Cadena & operator=(const Cadena &otra);  // asignacion

    int  longitud() const;
    char operator[](int i) const;
    char & operator[](int i);
    Cadena operator+(const Cadena &otra) const;
    bool   operator==(const Cadena &otra) const;
};
```

La batería de pruebas que hay que pasar, y qué detecta cada una:

```
Cadena a("hola");
Cadena b = a;          // constructor de copia
Cadena c;
c = a;                 // operador de asignacion
c = c;                 // autoasignacion: no debe romper nada
{
    Cadena d = a;      // se destruye al salir del bloque
}
assert(a.longitud() == 4);   // a debe seguir intacta
```

Prueba	Qué detecta si falla
`Cadena b = a;` y modificar `b`	copia superficial: cambia también `a`
Salir de un bloque con una copia viva	liberación doble al destruirse las dos
`c = c;`	falta la comprobación de autoasignación
Asignar dos veces al mismo objeto	fuga: no se liberó lo anterior
Valgrind al final	cualquiera de los cuatro

La comprobación con Valgrind es la que cierra la práctica, porque tres de esos cuatro errores pueden no manifestarse ejecutando.

6.4 Prácticas sobre el tema 4. Ficheros

- Leer un fichero de texto con campos separados y cargarlo en un vector de registros.
- Escribir el resultado en otro fichero, comprobando la apertura en los dos casos.
- El mismo programa con fichero binario, y comparar tamaños y tiempos.
- Acceso directo: modificar el registro i de un fichero binario sin leer los demás.

```
ifstream f("alumnos.csv");
if (!f) { cerr << "No se pudo abrir" << endl; return 1; }

string linea;
getline(f, linea);              // descartar la cabecera
while (getline(f, linea)) {
    istringstream campos(linea);
    string nombre, edadTexto, notaTexto;
    getline(campos, nombre,    ',');
    getline(campos, edadTexto, ',');
    getline(campos, notaTexto, ',');
    // convertir y guardar
}
```

Los dos errores que el guion busca:

- `while (!f.eof())`, que procesa el último elemento dos veces.
- No comprobar la apertura, con lo que un fichero inexistente produce cero registros y ningún mensaje.

6.5 Proyecto informático de programación

Reúne los cuatro temas y los cinco seminarios. La guía docente lo describe como un proyecto completo con análisis, diseño, implementación y documentación, y ese es el reparto del trabajo.

6.5.1 Lo que debe tener

Aspecto	Qué se espera
Estructura	módulos con su cabecera y su fuente
Construcción	`makefile` con dependencias correctas, incluidas las cabeceras

Aspecto	Qué se espera
Tipos propios	al menos una clase con gestión de memoria y la regla de los tres
Ficheros	persistencia de los datos, texto o binario
Errores	valores de retorno, aserciones y excepciones donde corresponda
Documentación	Doxygen sobre toda la interfaz pública
Pruebas	casos elegidos, incluidos los límite
Memoria	sin fugas, verificado con Valgrind

6.5.2 Estructura de trabajo

```
proyecto/
  include/   *.h
  src/       *.cpp
  obj/       *.o
  bin/       ejecutable
  doc/       Doxyfile y salida
  datos/     ficheros de prueba
  makefile
```

6.5.3 Orden de desarrollo

1. **Analizar**: qué datos hay y qué operaciones se piden.
2. **Diseñar** los tipos, y escribir sus cabeceras con la documentación completa antes de implementar nada.
3. **Implementar módulo a módulo**, probando cada uno por separado en cuanto compila.
4. **Integrar** y probar el conjunto.
5. **Pasar Valgrind** en cada entrega parcial, no al final.

El punto 5 es el que más tiempo ahorra. Una fuga introducida en la primera semana y detectada en la última obliga a revisar todo el código; detectada el mismo día, son dos líneas.

6.5.4 Antes de entregar

```
make clean && make                  # compila sin avisos desde cero
./bin/programa < casos/prueba1.txt  # con los casos elegidos
valgrind --leak-check=full ./bin/programa < casos/prueba1.txt
doxygen doc/Doxyfile                # la documentacion se genera sin errores
```

Y la revisión final: ningún aviso del compilador, ninguna variable global no constante, ningún `new` sin su `delete`, ninguna función sin documentar y ningún fichero abierto sin comprobar. Los guiones de estas prácticas y sus ejercicios están en Garrido Carrillo, 2017 y en Garrido Carrillo y Martínez-Baena, 2016; el planteamiento del proyecto, en Garrido Carrillo, 2016.

Bibliografía

Deitel, P., & Deitel, H. (2017). *C++: How to Program* (10.ª ed.). Pearson.

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

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

Garrido Carrillo, A. (2017). *Prácticas con C++: metodología de la programación* (2.ª ed.). Editorial Universidad de Granada.

Garrido Carrillo, A., & Martínez-Baena, J. (2016). *Introducción a la programación con C++: ejercicios*. Editorial Universidad de Granada.

Stroustrup, B. (2013). *The C++ Programming Language* (4.ª ed.). Addison-Wesley Professional.