

# UNIDAD 1: "INTRODUCCIÓN"

## 1.1 El modelo de objetos

Es un paradigma de desarrollo basado en:

- **Abstracción**
- **Encapsulamiento**
- **Modularidad**
- **Jerarquía**
- **Tipificación (tipos)**
- **Concurrencia**
- **Persistencia**

De este modelo se derivan tres niveles de aplicación:


- **Programación Orientada a Objetos (POO)**
- **Diseño Orientado a Objetos (DOO)**
- **Análisis Orientado a Objetos (AOO)**

---

### ◆ 1.1.1 Programación Orientada a Objetos (POO)

Es un **método de implementación** basado en:

- Programas organizados como **colecciones de objetos**.
- Cada objeto es una **instancia de una clase**.
- Las clases se **organizan jerárquicamente por herencia**.

 Un lenguaje se considera orientado a objetos si soporta:

- Objetos con **estado oculto** (encapsulado).

- **Interfaz de operaciones con nombre.**
- **Tipos asociados** (clases).
- **Herencia** de atributos y comportamiento.

🛑 Si falta alguno de estos, el lenguaje **no es verdaderamente OOP**, aunque use estructuras similares (como tipos abstractos).

---

### ♦ 1.1.2 Diseño Orientado a Objetos (DOO)

Es un **método de diseño** que:

- Utiliza la **descomposición en objetos y clases**.
- Emplea **notaciones específicas** para representar los modelos del sistema.

📌 En esta materia se usa **UML (Unified Modeling Language)** para modelar visualmente.

---

### ♦ 1.1.3 Análisis Orientado a Objetos (AOO)

Es un método para:

- Analizar los **requisitos del sistema** desde la perspectiva de objetos y clases del **dominio del problema**.

🔄 El análisis OO se convierte en la base del diseño OO → que luego se transforma en código con POO.

---

## 📌 1.2 Elementos del modelo objeto

### ♦ Abstracción

- Se enfoca en **las características esenciales** de un objeto.
- Define **fronteras conceptuales** claras.
- Se **oculta cómo funciona internamente**, solo importa **qué hace**.

#### ♦ Encapsulamiento

- Junta en una unidad la **estructura y el comportamiento** de un objeto.
- **Separa interfaz e implementación.**
- Los detalles internos están ocultos; al cliente solo le interesa lo que puede hacer el objeto.

 A estos detalles internos se los llama los “**secretos de la abstracción**”.

#### ♦ Modularidad


- División del sistema en **módulos cohesivos y débilmente acoplados**.
- Ayuda a reducir la complejidad.
- Cada módulo tiene fronteras bien definidas.

#### ♦ Jerarquía

- Organización de clases (**jerarquía de clases**) y de objetos (**jerarquía de partes**).
- Permite clasificar por nivel de generalización.

#### ♦ Tipos (tipificación)

- Define el **tipo de cada objeto**.
- Asegura que no se mezclen objetos incompatibles.
- Lenguajes **fuertemente tipados** detectan errores en tiempo de compilación (ventaja).

 Se introduce el **polimorfismo**: mismo nombre, diferentes comportamientos en clases relacionadas por herencia.

#### ♦ Concurrencia

- Permite que **varios objetos actúen al mismo tiempo**.
- Ej: threads en C++, tareas en Ada, procesos en Smalltalk.

#### ♦ Persistencia

- Un objeto **persiste en el tiempo o en el espacio**, más allá de su creador.
  - Lograda con **bases de datos**, o almacenamiento en variables globales, heaps, etc.
  - En sistemas modernos: se usa una **capa orientada a objetos sobre una base de datos relacional** (ej: Hibernate).
- 



## PREGUNTAS TIPO PARCIAL – UNIDAD 1

---

❶ ¿Cuál de los siguientes NO es un elemento fundamental del modelo de objetos?

- A. Encapsulamiento
- B. Modularidad
- C. Normalización
- D. Abstracción



**Respuesta correcta: C**

**Justificación:** La normalización pertenece al modelado de bases de datos, no al modelo de objetos.

---

❷ ¿Qué característica define mejor al encapsulamiento?

- A. Permite la herencia de atributos
- B. Separa la interfaz de la implementación
- C. Clasifica objetos por jerarquías
- D. Limita la ejecución concurrente



**Respuesta correcta: B**

**Justificación:** El encapsulamiento **oculta los detalles internos** de un objeto, y **expone solo la interfaz externa**.

---

❸ En la Programación Orientada a Objetos, ¿cuál es una condición necesaria para considerar a un lenguaje realmente "orientado a objetos"?

- A. Que tenga estructuras condicionales
- B. Que soporte bases de datos
- C. Que permita definir tipos primitivos
- D. Que tenga clases, herencia y objetos con estado oculto

✓ **Respuesta correcta:** D

**Justificación:** Si no se cumplen estos tres elementos, el lenguaje **no es verdaderamente OOP**, aunque tenga estructuras parecidas.

---

④ **¿Cuál de las siguientes afirmaciones sobre el diseño orientado a objetos es correcta?**

- A. Rompe con los modelos tradicionales y es revolucionario
- B. No se basa en el análisis orientado a objetos
- C. Utiliza UML para representar el sistema
- D. Es una técnica de implementación, no de diseño

✓ **Respuesta correcta:** C

**Justificación:** El diseño OO **utiliza UML** como notación para representar tanto modelos estáticos como dinámicos del sistema.

---

⑤ **¿Qué ventaja tiene un lenguaje con tipificación fuerte?**

- A. Evita la necesidad de declarar atributos
- B. No permite polimorfismo
- C. Detecta errores de tipo en tiempo de compilación
- D. No requiere compilar para ejecutar el código

✓ **Respuesta correcta:** C


**Justificación:** La **comprobación de tipos estricta** permite detectar errores temprano, lo que reduce fallos en ejecución.

## UNIDAD 2: "ELEMENTOS BÁSICOS Y CONCEPTOS"

### 2.1 Concepto o Tipo de Objeto

- Un **concepto** es una **idea o noción abstracta** que usamos para reconocer y clasificar cosas.
- Se representa mediante:
  - Su **nombre**
  - Su **definición** (comprensión)
  - Y el **conjunto de objetos** que lo representan (extensión)

 Si falta el nombre o la definición → es error de análisis.

 Si faltan objetos → puede ser válido (el objeto aún no existe).

#### Variaciones de conceptos:

- **Sinónimos (alias):** Dos nombres distintos para el mismo concepto.
  - **Homónimos:** Mismo nombre para diferentes conceptos.
- 

### 2.1 Objetos

Distintas definiciones de "objeto" según autores:

- **Encapsulación de estado + operaciones.**
- **Instancia de una clase.**
- **Entidad con identidad, estado y comportamiento.**
- **Corresponde a cosas del dominio del problema** (pueden ser tangibles o conceptuales).

 Todos coinciden en que:

- Un objeto tiene **estado, identidad y comportamiento**.

- Está asociado a una **clase o concepto** que lo representa.
  - Se comunica por medio de **mensajes/métodos**.
- 

## 2.2 Relaciones entre objetos

Una **relación** es un **vínculo entre objetos**. Se representa con una **línea entre clases u objetos**.

### ♦ 2.2.1 Asociación

- Conecta objetos de distintas clases (o de la misma).
- Ejemplo: Alumno – cursa – Materia.
- Puede convertirse en **Clase Relación** si tiene atributos propios (ej: Contrato laboral entre Empresa y Empleado).

### ♦ 2.2.2 Adornos de asociación:

- **Nombre**: describe la relación.
- **Dirección**: indica sentido (muy útil si hay ambigüedad).
- **Rol**: especifica el papel que juega cada clase en la relación.

Ejemplo: Vehículo tiene dos relaciones con Localidad: “fabricado en” y “radicado en”.

### ♦ 2.2.2.3 Multiplicidad (Cardinalidad)

- Indica **cuántos objetos pueden estar relacionados** con otro.
- Ejemplos:
  - 1 (uno)
  - 0..1 (cero o uno)
  - \* (muchos)
  - 1..5 (entre uno y cinco)

- ¡Es fundamental en diseño! Un error en la cardinalidad afecta directamente la lógica del sistema y el modelo de base de datos.
- 

### 2.2.3 Agregación

- Relación “**todo–parte**” entre objetos.
  - Las partes pueden existir de forma **independiente del todo**.
  - Ejemplo: Un velero está compuesto por velas, mástil, casco.
- 

### 2.2.4 Composición

- Relación más fuerte que la agregación.
  - Las partes **no pueden existir sin el todo**.
  - Ejemplo: Las hojas de un documento digital, si se borra el documento, se borran todas sus páginas.
- 

## 2.3 Clasificación

Proceso mediante el cual un objeto es asociado a un concepto.  
Permite **organizar** el conocimiento y la estructura de un sistema.

### ♦ 2.3.1 Herencia / Generalización

- Relación entre un **supertipo** (más general) y **subtipos** (más específicos).
- El subtipo **hereda atributos y comportamiento** del supertipo.
- Ejemplo: **Animal** → **Perro**, **Gato**, **Pez**.

### ♦ Sinónimos importantes:

- Supertipo = clase padre = clase base
- Subtipo = clase hija = clase derivada



## ♦ Particiones

- División de un tipo de objeto en varios subtipos.

Tipos de particiones:

- **Exclusivas:** un objeto pertenece solo a un subtipo (Ej: Solvente o No Solvente).
  - **Completas:** todos los subtipos están definidos.
  - **Incompletas:** hay subtipos no especificados (y puede estar bien según el modelo).
- 

## 2.4 Atributos, Estado, Eventos y Métodos

### ♦ 2.4.1 Atributos

- Características del objeto.
- Tienen: visibilidad, nombre, tipo, multiplicidad, valor por defecto, restricciones.

Tipos:

- **Primitivos** (int, float, bool, char)
- **No primitivos / complejos** (otros objetos)

Visibilidad:

- **+** público
- **-** privado
- **#** protegido

Puede haber atributos **derivados** (/) que se calculan a partir de otros.

### ♦ 2.4.2 Estado

- Es la “foto” del objeto en un momento dado: sus atributos, relaciones, y asociaciones.

### ♦ 2.4.3 Eventos

- Causan un cambio de estado.
- Tipos de eventos:
  - **Internos:** el cambio queda dentro del dominio.
  - **Externos:** el cambio ocurre fuera del sistema.
  - **Temporales:** provocados por el paso del tiempo (temporizador, reloj, etc).

Ejemplos:

- Evento de creación: se instancia un objeto.
- Evento de terminación: el objeto desaparece.

También se usan:

- **Pre-estado / Post-estado** → estado antes y después del evento.

### ♦ 2.4.4 Métodos (u operaciones)

- Implementan eventos.
- Son funciones asociadas a un objeto.
- Tienen parámetros de entrada, valores de salida, precondiciones y postcondiciones.

Notación UML:

+ nombre(parámetros): tipoRetorno



## PREGUNTAS TIPO PARCIAL – UNIDAD 2

---

❶ ¿Cuál de las siguientes afirmaciones sobre objetos es correcta?

- A. Un objeto no puede tener estado si pertenece a una clase
- B. Un objeto es una instancia de una clase

- C. Un objeto no puede tener operaciones propias
- D. Un objeto es lo mismo que una función

✓ **Respuesta correcta:** B

**Justificación:** En OOP, un objeto **siempre es una instancia de una clase**.

---

② **¿Cuál es la diferencia principal entre agregación y composición?**

- A. La agregación es jerárquica y la composición no
- B. En la composición, las partes no existen sin el todo
- C. En la agregación, las partes desaparecen con el todo
- D. Son equivalentes, solo cambia el nombre en UML

✓ **Respuesta correcta:** B

**Justificación:** La **composición** implica dependencia total de las partes respecto del todo.

---

③ **¿Qué representa la cardinalidad 1..5 en una asociación?**

- A. Que cada objeto está asociado exactamente con 1 o 5 objetos
- B. Que puede estar asociado a 0, 1 o más objetos
- C. Que debe estar asociado a entre 1 y 5 objetos
- D. Que puede haber como máximo 5 clases en la asociación

✓ **Respuesta correcta:** C

**Justificación:** La cardinalidad **1..5** indica un **rango mínimo y máximo de instancias asociadas**.

---

④ **¿Cuál de los siguientes elementos es típico de un atributo UML derivado?**

- A. Está subrayado
- B. Lleva el símbolo / antes del nombre
- C. Se declara como #
- D. Tiene un tipo compuesto por defecto

✓ **Respuesta correcta:** B

**Justificación:** En UML, los **atributos derivados** se indican anteponiendo **/** al nombre.

---

⑤ **¿Qué define una precondición en un método?**

- A. El resultado esperado al finalizar el método
- B. El valor de retorno del método

- C. La visibilidad del método
- D. Las condiciones que deben cumplirse antes de ejecutarlo

✓ **Respuesta correcta: D**

**Justificación:** Una **precondición** determina **qué requisitos deben cumplirse para ejecutar correctamente un método**.

## **UNIDAD 3: “LA CLASE”**

### **3.1 La clase como estructura**

Una **clase** es un **Tipo de Dato Abstracto (TDA)**, que permite:

- Modularidad
- Flexibilidad
- Reutilización

**Es el concepto fundamental del paradigma orientado a objetos.**

Más importante aún que el objeto en sí, porque:

 El objeto **no existe sin una clase** que lo describa.

---

#### **¿Qué es una clase?**

- Es una **estructura que define**:
  - Un conjunto de datos (atributos)
  - Operaciones (métodos)
  - Reglas de visibilidad y herencia

 La clase es un **molde**. El objeto es una **instancia concreta** del molde.

---

### ♦ Ejemplo: Clase como molde

Supongamos una clase **Libro**. Las instancias serían:

- Libro A (nuevo)
- Libro B (usado, con anotaciones)
- Libro C (con código de biblioteca)

Todos derivan del mismo **molde Libro**, pero cada instancia tiene **sus propias características individuales** (estado, identidad, comportamiento).

---

### ♦ Instancia vs Clase

Clase	Objeto (instancia)
Definición estática	Estructura dinámica en ejecución
Existe en el código fuente	Existe en tiempo de ejecución
Modelo general	Ejemplo concreto del modelo

---

#### 3.1.1 El molde y la instancia

- El molde es la **descripción general** del objeto.
  - La instancia es la **materialización específica**.
  - **Clase:** descripción que escribís como programador.
  - **Objeto:** es creado a partir de esa clase en memoria, con sus datos.
- 

#### 3.1.2 Clase como módulo y tipo

- **Módulo:** La clase actúa como una **unidad de código** con:
  - Interfaz pública (métodos accesibles)

- Implementación interna (datos y lógica)
  - **Tipo:** Cada clase es un **tipo de dato**, lo que permite:
    - Declarar variables
    - Definir restricciones de herencia
    - Control de tipos (polimorfismo)
- 

### 3.1.3 Sistema de tipos uniforme

- En lenguajes orientados a objetos **todo es un objeto**, incluso clases internas o primitivas (en algunos casos).
  - La clase proporciona un sistema de tipos **coherente y extensible**.
  - Posibilita el uso de **polimorfismo** y la reutilización por herencia.
- 

### 3.2 La estructura **class** en programación

Lenguajes como C++, Java, Python o Smalltalk utilizan **class** para declarar clases.

Una clase define:

- Atributos (estado)
- Métodos (comportamiento)
- Reglas de acceso
- Relaciones (herencia, composición, etc.)

#### ♦ Ejemplo en C++:

cpp

CopyEdit

```
class Persona {  
private:  
    string nombre;  
    int edad;
```

```
public:
    Persona(string nom, int ed);
    void saludar();
};
```

#### ♦ Clientes y proveedores

- **Cliente:** clase que **usa** otra clase.
- **Proveedor:** clase que **es usada** por otra.

Relación muy común: un objeto necesita otro para cumplir su funcionalidad.

---

### 3.3 El objeto como estructura dinámica

- Un objeto **ocupa memoria** y mantiene su propio estado.
- Se puede **crear, modificar, eliminar** durante la ejecución del programa.

#### ♦ 3.3.1 Manipulando objetos y referencias

- En lenguajes como C++ y Java, los objetos **se acceden mediante referencias o punteros**.
- Es posible:
  - Tener referencias nulas
  - Hacer alias (dos variables apuntando al mismo objeto)
  - Crear estructuras complejas (auto-referencias, grafos)

👁👁 La gestión de memoria es **crítica** en la creación y destrucción de objetos (constructores, destructores).

---



## PREGUNTAS TIPO PARCIAL – UNIDAD 3

---

❶ ¿Cuál de las siguientes afirmaciones es verdadera sobre la relación entre clase y objeto?

- A. Un objeto contiene varias clases
- B. Una clase es una instancia de un objeto
- C. Un objeto es una instancia de una clase
- D. La clase se crea en tiempo de ejecución

☒ **Respuesta correcta:** C

**Justificación:** Una clase es un molde, y los objetos son instancias derivadas de esa clase.

---

**2 ¿Qué representa mejor a una clase en términos de programación?**

- A. Una interfaz de hardware
- B. Un tipo de dato primitivo
- C. Un módulo con datos y comportamiento
- D. Un algoritmo de ordenamiento

☒ **Respuesta correcta:** C

**Justificación:** Una clase encapsula datos (atributos) y comportamiento (métodos), funcionando como módulo de software.

---

**3 ¿Qué significa que una clase actúe como tipo?**

- A. Que solo puede contener variables enteras
- B. Que no puede ser utilizada para crear objetos
- C. Que define un conjunto de estructuras que pueden ser instanciadas
- D. Que solo sirve para declarar funciones estáticas

☒ **Respuesta correcta:** C

**Justificación:** Una clase como tipo define un conjunto de objetos posibles, y se usa como base para crear instancias.

---

**4 ¿Qué se entiende por “cliente” en una relación cliente-proveedor entre clases?**

- A. Es la clase que hereda de otra
- B. Es la clase que utiliza servicios de otra clase
- C. Es la clase que implementa una interfaz
- D. Es la clase que no tiene atributos

☒ **Respuesta correcta:** B

**Justificación:** El cliente **depende** del proveedor para obtener funcionalidad.

---

**5 ¿Cuál de estas afirmaciones describe mejor el rol de un objeto durante la ejecución de un programa?**



- A. Es un componente estático sin comportamiento
- B. Es la descripción del programa en papel
- C. Es una entidad dinámica con atributos y comportamiento propio
- D. Es un tipo de función recursiva

✓ **Respuesta correcta:** C

**Justificación:** Un objeto vive en memoria, tiene estado, identidad, y puede ejecutar métodos.

## **UNIDAD 4: "TÉCNICAS DE ABSTRACCIÓN"**


### **4.1 Herencia**

La **herencia** es una **técnica de abstracción** que permite:

- Reutilizar código (atributos y métodos) de una clase base (superclase)
- Evitar redundancia
- Crear jerarquías lógicas de especialización

#### ♦ **Características:**

- La clase derivada **hereda** todos los miembros (visibles) de la clase base.
- Se pueden **agregar** atributos y métodos propios en la subclase.
- Se pueden **redefinir** métodos (override).

 El subtipo es una especialización del supertipo.  
Ejemplo: **Empleado** → **Ejecutivo** hereda de **Empleado**.

#### ♦ **Ventajas:**

- Reutilización
- Organización jerárquica del conocimiento
- Polimorfismo

♦ **En C++:**

```
cpp
CopyEdit
class Empleado {
public:
    void trabajar();
};

class Ejecutivo : public Empleado {
public:
    void tomarDecisiones();
};
```

`Ejecutivo` puede usar `trabajar()` porque lo hereda de `Empleado`.

---

## 4.2 Herencia múltiple

- En algunos lenguajes (como C++), una clase puede heredar de **más de una clase base**.

```
cpp
CopyEdit
class Volador {
public:
    void volar();
};

class Nadador {
public:
    void nadar();
};

class Pato : public Volador, public Nadador {};
```

### Problemas comunes:

- Ambigüedad: Si dos clases tienen métodos con el mismo nombre
- Orden de construcción y destrucción

✖ Requiere **cuidadoso diseño** para evitar conflictos de herencia (como el "problema del diamante").

---

## 📌 4.3 Clases abstractas

Una **clase abstracta** es una clase que:

- **No se puede instanciar**
- Se usa como **modelo base**
- Puede tener métodos **puros** (sin implementación)

### ♦ Método puro en C++:

```
cpp
CopyEdit
class Figura {
public:
    virtual void dibujar() = 0; // método puro
};
```

Cualquier clase que tenga al menos un método puro se considera **abstracta**.

### ♦ Objetivo:

- **Obligar a las subclases** a implementar ciertos métodos.
- Representar conceptos generales que **no tienen sentido concreto por sí mismos**.

Ejemplo: **Figura** no tiene sentido instanciarla directamente, pero sí **Circulo** o **Rectángulo**.

---

## 📌 4.4 Interfaces

Una **interfaz** es una colección de métodos **sin implementación** que una clase debe cumplir.

### ♦ Diferencias con clases abstractas:

- Una interfaz **no tiene atributos**, solo métodos.

- En lenguajes como Java o C#, las interfaces se declaran explícitamente.
- En C++, no existen como tal, pero se **simulan** usando clases abstractas con solo métodos puros.

```
cpp
CopyEdit
class Imprimible {
public:
    virtual void imprimir() = 0; // interfaz simulada
};
```

#### ♦ Ventajas:

- **Desacoplamiento** entre lo que se hace y cómo se hace.
- Permiten el **polimorfismo** sin herencia directa.
- Una clase puede implementar varias interfaces (incluso en lenguajes sin herencia múltiple de clases).

---

### 4.4.1 Referencias a interfaces

- En tiempo de ejecución, se puede usar una referencia del tipo interfaz para apuntar a cualquier objeto que la implemente.

```
cpp
CopyEdit
Imprimible* doc = new Factura();
doc->imprimir();
```

Esto es **polimorfismo** puro: la implementación real se resuelve dinámicamente.

---

### 4.4.2 Herencia de interfaces

- Una interfaz puede heredar de otra interfaz.
- Esto permite **composición** de capacidades.

Ejemplo: `Imprimible` y `Exportable` → `DocumentoExportable` puede heredar de ambas.

---



## PREGUNTAS TIPO PARCIAL – UNIDAD 4

---

### ❶ ¿Qué característica define a una clase abstracta?

- A. Puede tener métodos públicos y privados
- B. No puede ser heredada por otras clases
- C. No se puede instanciar y contiene al menos un método puro
- D. Tiene atributos privados y métodos sobrecargados

✅ **Respuesta correcta:** C

**Justificación:** Una clase abstracta no se puede instanciar y debe contener al menos un método declarado como puro (= 0 en C++).

---

### ❷ ¿Cuál es una diferencia clave entre clase abstracta e interfaz en C++?

- A. Las clases abstractas tienen más prioridad en ejecución
- B. Las interfaces no pueden ser utilizadas en C++
- C. Las interfaces solo contienen métodos puros, sin atributos
- D. Las clases abstractas no pueden tener métodos implementados

✅ **Respuesta correcta:** C

**Justificación:** En C++, las interfaces se simulan mediante clases abstractas **sin atributos** y con **solo métodos puros**.

---

### ❸ ¿Qué problema puede aparecer en la herencia múltiple?

- A. Falta de encapsulamiento
- B. Sobrecarga de operadores
- C. Ambigüedad por métodos con el mismo nombre
- D. Imposibilidad de reutilizar código

✅ **Respuesta correcta:** C

**Justificación:** En la herencia múltiple, **dos clases base pueden tener métodos con el mismo nombre**, generando ambigüedad.

---

### ❹ ¿Qué se logra utilizando interfaces en el diseño orientado a objetos?

- A. Reutilizar implementaciones internas
- B. Heredar el estado de la clase padre
- C. Separar la definición de la implementación
- D. Aumentar el acoplamiento entre clases

✓ **Respuesta correcta:** C

**Justificación:** Las interfaces **permiten definir el "qué" sin definir el "cómo"**, logrando **desacoplamiento y flexibilidad**.

---

⑤ **¿Cuál de las siguientes afirmaciones es verdadera sobre polimorfismo?**

- A. Se refiere a la ocultación de información dentro de una clase
- B. Permite usar múltiples clases como si fueran una sola interfaz común
- C. Solo se aplica cuando hay herencia múltiple
- D. Impide sobrescribir métodos en clases derivadas

✓ **Respuesta correcta:** B

**Justificación:** El **polimorfismo** permite que un mismo mensaje (método) sea interpretado de diferentes formas según la clase concreta del objeto que lo recibe.

## **UNIDAD 5: "SINTAXIS DE C++"**

Esta unidad tiene como objetivo introducir las **bases sintácticas de C++** y cómo se traduce el diseño orientado a objetos en código real.

---

### **5.1 Estructura de un programa en C++**

Un programa típico en C++ tiene esta estructura mínima:

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

int main() {
    // cuerpo del programa
    return 0;
}
```

### Componentes principales:

- `#include<iostream>` → incluye la librería estándar de entrada/salida.
  - `using namespace std;` → evita escribir `std::` antes de objetos como `cout`, `cin`.
  - `main()` → punto de inicio del programa.
- 

## 5.2 Compilación y ejecución

El proceso habitual es:

1. Escribís el código (`.cpp`)
  2. Lo compilás con un **compilador C++** (ej: g++, clang, Code::Blocks)
  3. Se genera un archivo ejecutable que se puede correr
- 

## 5.3 Variables

Las variables en C++ deben declararse antes de ser usadas.

```
cpp
CopyEdit
int edad = 25;
float promedio = 8.75;
```

Tipos comunes:

- `int` → enteros
- `float`, `double` → números con decimales
- `char` → caracteres
- `bool` → booleanos (`true`, `false`)
- `string` → texto (requiere `#include<string>`)

---

### 5.3.1 Estructuras de control

Permiten modificar el flujo del programa:

- **Condicionales:**

```
cpp
CopyEdit
if (condición) { ... } else { ... }
```

- **Bucles:**

```
cpp
CopyEdit
for (...) {...}
while (...) {...}
do {...} while (...);
```

- **Switch:**

```
cpp
CopyEdit
switch (var) {
    case 1: break;
    ...
}
```

---

### 5.3.2 Funciones

Bloques reutilizables de código que pueden recibir parámetros y devolver resultados.

```
cpp
CopyEdit
int sumar(int a, int b) {
    return a + b;
}
```



Toda función **debe declarar su tipo de retorno**. Si no devuelve nada, se usa `void`.

---

## 5.4 Objetos

En C++, los objetos se crean a partir de clases definidas por el programador.

```
cpp
CopyEdit
class Persona {
public:
    string nombre;
    int edad;

    void saludar() {
        cout << "Hola, soy " << nombre << endl;
    }
};
```

Luego se puede usar así:

```
cpp
CopyEdit
Persona p;
p.nombre = "Lucas";
p.saludar();
```

---

### 5.4.1 Las clases

Una clase define:

- **Atributos** (datos)
  - **Métodos** (funciones)
  - **Reglas de visibilidad:** `public`, `private`, `protected`
- 

## 5.5 Encapsulamiento

Separación entre **interfaz pública** (lo que se puede usar) y **implementación privada** (lo que se oculta).

---

### 5.5.1 Permisos públicos y privados

- **public**: accesible desde fuera de la clase.
  - **private**: solo accesible desde dentro.
  - **protected**: accesible solo desde la clase o subclases.
- 

### 5.5.2 Métodos

Funciones que actúan sobre los atributos del objeto. Pueden recibir parámetros y retornar valores.

---

### 5.5.3 Constructores y Destructores

- **Constructor**: método especial que se ejecuta al crear un objeto.

```
cpp
CopyEdit
class Persona {
public:
    Persona() {
        cout << "Objeto creado" << endl;
    }
};
```

- **Destructor**: método especial que se ejecuta al destruir el objeto.

```
cpp
CopyEdit
~Persona() {
    cout << "Objeto destruido" << endl;
}
```

---

## 5.6 Manejo de memoria

En C++, podés gestionar memoria **dinámicamente**:

```
cpp
CopyEdit
int* p = new int;      // asignación
*p = 10;
delete p;              // liberación
```

Se requiere cuidado para **evitar fugas de memoria** (memory leaks).

---

### 5.6.1 Notas sobre punteros

- Un puntero almacena **la dirección de memoria** de otra variable.
- El operador `*` se usa para **desreferenciar** (acceder al contenido).
- El operador `&` se usa para **obtener la dirección**.

```
cpp
CopyEdit
int x = 5;
int* px = &x;

cout << *px; // imprime 5
```



## PREGUNTAS TIPO PARCIAL – UNIDAD 5

---

❶ ¿Cuál es la función principal de un constructor en C++?

- A. Destruir objetos una vez que dejan de usarse
- B. Crear una clase hija a partir de una clase base
- C. Inicializar objetos al momento de su creación
- D. Declarar variables dentro de una clase

✓ **Respuesta correcta:** C

**Justificación:** El constructor **se ejecuta automáticamente** al crear un objeto e inicializa sus valores.

---

② ¿Qué palabra clave se utiliza para liberar memoria reservada con **new**?

- A. remove
- B. destroy
- C. delete
- D. free

✓ **Respuesta correcta:** C

**Justificación:** En C++, la memoria reservada con **new** se libera con **delete**.

---

③ ¿Cuál de estas opciones representa mejor una función en C++?

- A. Una estructura que contiene métodos y variables
- B. Un conjunto de instrucciones que se ejecutan automáticamente
- C. Un bloque de código reutilizable con nombre, parámetros y valor de retorno
- D. Una clase con herencia múltiple

✓ **Respuesta correcta:** C

**Justificación:** Una función en C++ puede recibir parámetros, tiene un tipo de retorno y se llama por su nombre.

---

④ ¿Qué operador se utiliza para acceder al valor de una dirección apuntada por un puntero?

- A. &
- B. ->
- C. %
- D. \*

✓ **Respuesta correcta:** D

**Justificación:** El operador **\*** permite **desreferenciar** el puntero y acceder al contenido de la dirección.

---

⑤ ¿Qué ocurre si olvidás liberar memoria reservada con **new**?

- A. El programa no compila
- B. Se produce una fuga de memoria (memory leak)

- C. El sistema libera automáticamente la memoria
- D. Se borra el contenido del puntero

✓ **Respuesta correcta:** B

**Justificación:** En C++, **el programador es responsable** de liberar memoria; si no lo hace, se acumulan fugas de memoria.

## **UNIDAD 6: “ALGUNOS TIPOS DE DATOS FUNDAMENTALES”**

### **6.1 Precedencia y asociatividad de operadores**

En C++, **los operadores tienen un orden de evaluación** cuando se combinan en una expresión.

**Ejemplo:**

cpp

CopyEdit

```
int x = 5 + 3 * 2;
```

Primero se ejecuta  $3 * 2$ , luego se suma 5. Resultado: 11.

 **Reglas clave:**

- **Precedencia:** qué operador se ejecuta primero.
- **Asociatividad:** orden de evaluación cuando hay operadores con la misma precedencia.
  - Ej: la mayoría de los operadores aritméticos son **asociativos por izquierda**.

 Tabla resumida de precedencia (de mayor a menor):

1.  $()$  paréntesis
2.  $*, /, \%$
3.  $+, -$
4.  $==, !=$

5. `&&, ||`

6. `=, +=, -=` (asignación: asociatividad **derecha a izquierda**)

---

## 6.2 Tipos de datos

En C++, existen **tipos primitivos** que sirven de base para declarar variables. Estos se dividen en:


---

### ◆ 6.2.1 Números enteros (**int**)

- Tipo base para números sin decimales.
- Tamaño habitual: **32 bits**
- Rango: de **-2.147.483.648 a 2.147.483.647**

Variantes modificadas por tamaño o signo:

Tipo	Tamaño	Rango aproximado
<code>short int</code>	16 bits	-32.768 a 32.767
<code>unsigned short int</code>	16 bits	0 a 65.535
<code>unsigned int</code>	32 bits	0 a 4.294.967.295
<code>long long int</code>	64 bits	±9 cuatrillones (2 <sup>63</sup> aprox)
<code>unsigned long long</code>	64 bits	0 a 2 <sup>64</sup>

 **unsigned** = solo positivos → permite mayor rango superior.

---

### ◆ 6.2.2 Números flotantes (**float, double, long double**)

Usados para representar **números con decimales**.

Tipo	Tamaño	Precisión	Rango aproximado
<code>float</code>	32 bits	hasta 6 cifras decimales	$\sim \pm 3.4 \times 10^{38}$
<code>double</code>	64 bits	hasta 15 cifras	$\sim \pm 1.7 \times 10^{308}$
<code>long double</code>	96 bits	hasta 18 cifras	$\sim \pm 1.2 \times 10^{4932}$

⚠ La precisión varía según la arquitectura del compilador.

### 📌 6.2.3 Formateando la salida (`iomanip`)

Para controlar cómo se muestran los datos por `cout`, se usa la librería `<iomanip>`.

Ejemplos útiles:

```
cpp
CopyEdit
#include <iomanip>
cout << fixed << setprecision(2) << 3.14159; // imprime 3.14
```

Otros formatos:

- `setw(n)` → fija el ancho
- `setfill('x')` → rellena con 'x'
- `left, right` → alineación

### 📌 6.3 Usando caracteres (`char`)

- Tipo `char` → representa un solo **carácter ASCII**

```
cpp
CopyEdit
char letra = 'A';
```

- Internamente es un número entero entre 0 y 255.

⚠ El valor 'A' equivale a **65** en ASCII.

Se puede operar con ellos como si fueran enteros:

```
cpp
CopyEdit
cout << (char)(letra + 1); // imprime 'B'
```

---

## 📌 6.4 El tipo **auto**

Permite **inferir automáticamente** el tipo de una variable según el valor asignado.

```
cpp
CopyEdit
auto x = 42;           // int
auto y = 3.14;         // double
auto nombre = "Ana";  // const char*
```

🔒 El tipo se fija en el momento de compilación. No es como JavaScript.

---

## 📌 6.5 Enumeraciones (**enum**)

Permiten definir **conjuntos de valores simbólicos**:

```
cpp
CopyEdit
enum Color { ROJO, VERDE, AZUL };
Color c = VERDE;
```

- Los valores por defecto comienzan en 0.
- Se pueden asignar valores explícitos:

```
cpp
CopyEdit
enum Estado { ACTIVO = 1, INACTIVO = 0 };
```

---



## 6.6 Alias para tipos (**typedef**, **using**)


Permiten crear **nombres personalizados** para tipos complejos.

cpp

CopyEdit

```
typedef unsigned int Edad;
```

```
using Precio = float;
```

 **using** es más moderno y preferido en C++11 en adelante.

---



## PREGUNTAS TIPO PARCIAL – UNIDAD 6

---

❶ ¿Cuál es el tipo de dato más adecuado para almacenar un número con 15 cifras decimales de precisión?

- A. **float**
- B. **long**
- C. **double**
- D. **char**

✅ **Respuesta correcta:** C

**Justificación:** **double** permite almacenar números con **hasta 15 cifras decimales**, a diferencia de **float** (6 cifras).

---

❷ ¿Qué valor representa el carácter '**A**' en código ASCII?

- A. 90
- B. 65
- C. 33
- D. 97

✅ **Respuesta correcta:** B

**Justificación:** En la tabla ASCII, '**A**' corresponde al valor **65**.

---

❸ ¿Cuál es el resultado de esta expresión en C++?:

cpp

CopyEdit

```
int x = 5 + 3 * 2;
```

- A. 16
- B. 11
- C. 10
- D. 13

✓ **Respuesta correcta:** B

**Justificación:** Por precedencia de operadores, primero se evalúa  $3 * 2 = 6$ , luego  $5 + 6 = 11$ .

---

④ ¿Qué palabra clave permite que el compilador determine automáticamente el tipo de una variable?

- A. `define`
- B. `auto`
- C. `typedef`
- D. `let`

✓ **Respuesta correcta:** B

**Justificación:** `auto` en C++ permite **inferir el tipo** según el valor asignado.

---

⑤ ¿Qué se logra al usar `setw(6)` en la salida por pantalla?

- A. Asignar una variable con valor 6
- B. Imprimir solo 6 caracteres
- C. Reservar 6 espacios de ancho para el dato
- D. Crear una tabla de 6 columnas

✓ **Respuesta correcta:** C

**Justificación:** `setw(n)` fija el **ancho mínimo** de impresión para el siguiente dato en la salida de `cout`.

## UNIDAD 7: "VARIABLES"

### 7.1 Precedencia y asociatividad de operadores (revisión)

Se recuerda el orden en que se evalúan los operadores.

#### Prioridades altas:

- `()` Paréntesis → siempre se evalúa primero
- `*`, `/`, `%` → multiplicación, división y módulo
- `+`, `-` → suma y resta

#### Asociatividad:


- **Izquierda a derecha:** la mayoría (suma, resta, comparación)
  - **Derecha a izquierda:** asignación (`=`, `+=`, `-=`), y operadores unarios (`++`, `--`)
- 

### 7.2 Tipos de datos (revisado y ampliado)

#### ♦ 7.2.1 Números enteros

- Se refuerza la tabla de enteros:

Tipo	Tamaño	Rango aproximado
<code>short int</code>	16 bits	-32.768 a 32.767
<code>unsigned short int</code>	16 bits	0 a 65.535
<code>int</code>	32 bits	±2.147.483.648
<code>unsigned int</code>	32 bits	0 a 4.294.967.295
<code>long long int</code>	64 bits	±9 cuatrillones (2 <sup>63</sup> aprox)
<code>unsigned long long int</code>	64 bits	0 a 2 <sup>64</sup>

 Todos estos pueden usarse con **auto** si se desea inferencia.

---

### 7.2.2 Números flotantes

- **float** → precisión simple (~6 cifras decimales)
- **double** → precisión doble (~15 cifras)
- **long double** → precisión extendida (~18 cifras)

Rangos similares a los vistos en la Unidad 6.

---

### 7.2.3 Formateando la salida (**iomanip**)

Revisión con **más ejemplos prácticos**:

```
cpp
CopyEdit
cout << setprecision(4) << 3.141592653; // imprime 3.142
cout << setw(10) << 123;                // deja espacios
cout << setfill('*') << setw(5) << 7;    // ****7
```

Útil para alinear columnas o imprimir reportes numéricos.

---

### 7.3 Usando caracteres (**char**)

- Se recuerda que **char** almacena **un único carácter** entre comillas simples.
- Se puede operar aritméticamente con ellos porque son números ASCII.

```
cpp
CopyEdit
char letra = 'A';
cout << letra + 1;    // 66
cout << (char)(letra + 1); // B
```

---

## 7.4 El tipo `auto`

Se refuerza el uso de `auto` como herramienta útil en C++ moderno:

```
cpp
CopyEdit
auto edad = 30;           // int
auto pi = 3.14;           // double
auto letra = 'X';         // char
```

👁👁 No es dinámico como en otros lenguajes: el tipo se  **fija en tiempo de compilación**.

---

## 7.5 Enumeraciones (`enum`)

Se repasa el uso de `enum` para **definir conjuntos de constantes simbólicas**:

```
cpp
CopyEdit
enum Dia { Lunes, Martes, Miercoles };
```

- Por defecto, `Lunes = 0`, `Martes = 1`, etc.
- Pueden asignarse valores personalizados:

```
cpp
CopyEdit
enum Estado { ACTIVO = 1, INACTIVO = 0 };
```

---

## 7.6 Alias para tipos

### `typedef`

Forma clásica de crear un alias de tipo:

```
cpp
CopyEdit
typedef unsigned int Edad;
Edad e = 23;
```

## using

Forma moderna (desde C++11):

```
cpp
CopyEdit
using Altura = float;
Altura h = 1.75;
```

✅ Mejora la **claridad del código** y evita escribir tipos complejos repetidamente.

---



## PREGUNTAS TIPO PARCIAL – UNIDAD 7

---

❶ ¿Qué resultado muestra esta expresión?

```
cpp
CopyEdit
int x = 10;
int y = 3;
int z = x / y;
```

- A. 3.333
- B. 3
- C. 0.3
- D. Error de compilación

✅ **Respuesta correcta:** B

**Justificación:** Como ambos operandos son `int`, la división es **entera** → el resultado es **3**.

---

❷ ¿Cuál es el valor ASCII correspondiente al carácter '**B**'?

- A. 66
- B. 65
- C. 98
- D. 64

✅ **Respuesta correcta:** A

**Justificación:** En ASCII, '**B**' tiene valor **66**. ('**A**' = 65, '**C**' = 67...)

---

③ ¿Qué ventaja ofrece el uso de **auto** en declaraciones?

- A. Hace el programa más lento pero más seguro
- B. Permite cambiar el tipo de la variable en tiempo de ejecución
- C. Permite al compilador inferir el tipo según el valor asignado
- D. Hace que todas las variables sean de tipo **int** por defecto

✓ **Respuesta correcta:** C

**Justificación:** **auto** permite **inferir automáticamente** el tipo de una variable según su inicialización.

---

④ ¿Qué significa **setfill('\*')** en una instrucción de salida?

- A. Llenar la memoria con asteriscos
- B. Cambiar el carácter de fondo del texto
- C. Usar el carácter **\*** para rellenar espacios al imprimir
- D. Convertir todos los números en formato hexadecimal

✓ **Respuesta correcta:** C

**Justificación:** **setfill('\*')** rellena los espacios generados por **setw()** con asteriscos en lugar de espacios en blanco.

---

⑤ ¿Cuál de estas líneas define un alias de tipo **unsigned int** llamado **Edad** usando la forma moderna?

- A. `typedef unsigned int Edad;`
- B. `alias Edad = unsigned int;`
- C. `using Edad = unsigned int;`
- D. `auto Edad = unsigned int;`

✓ **Respuesta correcta:** C

**Justificación:** En C++ moderno (desde C++11), **using Nombre = Tipo;** es la forma estándar de definir un alias.

## UNIDAD 8: "CADENAS DE CARACTERES"

### 8.1 El tipo `string`

El tipo `string` es una **clase** de la biblioteca estándar de C++ (`std`) que permite trabajar con **cadenas de texto** (secuencias de caracteres).

**Requiere incluir:**

```
cpp
CopyEdit
#include <string>
```

**Declaración básica:**

```
cpp
CopyEdit
string nombre = "Lucas";
```

**Ventajas frente a arreglos de `char`:**

- Gestión dinámica de memoria
  - Métodos integrados
  - Mayor seguridad
  - Más legible
- 

### 8.2 Inicialización y asignación

Hay varias formas de inicializar un `string`:

```
cpp
CopyEdit
string s1;                // string vacío
string s2 = "Hola";       // literal
string s3("Mundo");       // constructor
string s4 = s2 + " " + s3; // concatenación
```

También se pueden usar operadores de asignación:



```
cpp
CopyEdit
s1 = "Texto nuevo";
```

 La clase `string` sobrecarga los operadores `+`, `=`, `==`, etc.

---

## 8.3 Entrada y salida

**Salida (`cout`):**

```
cpp
CopyEdit
cout << s2;
```

**Entrada (`cin`):**

```
cpp
CopyEdit
cin >> s1;    // Lee hasta espacio
```

Para leer una **línea completa** (incluyendo espacios):

```
cpp
CopyEdit
getline(cin, s1);
```

 Recomendado usar `getline()` si se espera texto con espacios.

---

## 8.4 Tamaño de una cadena

```
cpp
CopyEdit
string palabra = "computadora";
cout << palabra.size();    // 11
```

Métodos equivalentes:

- `size()`

- `length()`

Ambos retornan la cantidad de caracteres.


---

## 8.5 Concatenación

cpp

CopyEdit

```
string nombre = "Lucas";  
string saludo = "Hola, " + nombre;
```

 Se puede concatenar usando `+` o `+=`:

cpp

CopyEdit

```
nombre += " Sellart";
```

---

## 8.6 Comparación

Los strings pueden compararse con:

cpp

CopyEdit


```
if (s1 == s2) { ... }  
if (s1 != s3) { ... }
```

También son posibles las comparaciones lexicográficas:

cpp

CopyEdit

```
if (s1 < s2) { ... } // según orden alfabético
```

 Los operadores `==`, `!=`, `<`, `>`, etc., están sobrecargados para `string`.

---

## 8.7 Acceso a caracteres individuales

cpp

CopyEdit

```
string palabra = "hola";
```

```
char letra = palabra[1];    // 'o'
```

🧠 Los índices empiezan en 0.

También podés usar `.at()`:

```
cpp
CopyEdit
char c = palabra.at(2);    // 'l'
```

⚠️ `.at()` lanza excepción si el índice es inválido.

---

## 📌 8.8 Métodos útiles

Algunos métodos comunes de `string`:

Método	Función
<code>size() / length()</code>	Cantidad de caracteres
<code>empty()</code>	Retorna <code>true</code> si el string está vacío
<code>at(i)</code>	Retorna el carácter en la posición <code>i</code>
<code>substr(pos, len)</code>	Subcadena desde <code>pos</code> con <code>len</code> caracteres
<code>find("txt")</code>	Posición de la primera ocurrencia de <code>"txt"</code>
<code>replace(pos, len, "nuevo")</code>	Reemplaza una parte del string

---

## 📌 8.9 Errores comunes

- No incluir `<string>` → error de tipo no reconocido
  - Usar `cin` para leer texto con espacios → solo lee hasta el primer espacio
  - Acceder fuera del rango del string → error en tiempo de ejecución (con `.at()`)
-



## PREGUNTAS TIPO PARCIAL – UNIDAD 8

---

❶ ¿Qué función se utiliza para leer una línea completa (con espacios) desde el teclado?

- A. `cin >>`
- B. `getline(cin, variable)`
- C. `string_input()`
- D. `readline()`

✅ Respuesta correcta: B

**Justificación:** `getline(cin, variable)` permite leer toda una línea de texto, incluidos los espacios.

---

❷ ¿Qué resultado imprime este código?

```
cpp
CopyEdit
string nombre = "Ana";
cout << nombre[1];
```

- A. A
- B. n
- C. a
- D. Error en tiempo de ejecución

✅ Respuesta correcta: B

**Justificación:** Los índices empiezan en 0, así que `nombre[1]` accede al segundo carácter, que es 'n'.

---

❸ ¿Qué método permite obtener una parte del string a partir de una posición y una cantidad de caracteres?

- A. `cut()`
- B. `extract()`
- C. `substr()`
- D. `section()`

✅ Respuesta correcta: C

**Justificación:** `substr(pos, len)` devuelve una subcadena a partir del índice `pos`, con `len` caracteres.

---

④ ¿Qué retorna el método `empty()` de un string?

- A. La cantidad de espacios en blanco
- B. Un string con espacio vacío
- C. `true` si el string está vacío
- D. `false` si contiene más de una palabra

✅ Respuesta correcta: C

**Justificación:** `empty()` verifica si el string no contiene caracteres → retorna `true` si está vacío.

---

⑤ ¿Qué sucede si se intenta acceder a un índice fuera del rango con `.at()`?

- A. El programa retorna un valor vacío
- B. El programa lanza una excepción
- C. Se imprime un carácter nulo
- D. El índice se ajusta automáticamente

✅ Respuesta correcta: B

**Justificación:** `.at()` verifica los límites y lanza una excepción si se intenta acceder a una posición no válida.

## UNIDAD 9: “LA CLASE VECTOR”

### 9.1 ¿Qué es un `vector`?

Un `vector` es una **estructura de datos secuencial** de la biblioteca estándar de C++ (`std::vector`) que:

- Permite almacenar múltiples elementos del mismo tipo
- Se **redimensiona automáticamente**
- Reemplaza al clásico arreglo `[ ]` (más flexible y seguro)

- 🔧 Pertenece a la librería `<vector>`
  - 📖 Se usa junto con `#include <vector>`
- 

## 📌 9.2 Declaración e inicialización

**Sintaxis básica:**

```
cpp
CopyEdit
vector<int> numeros;           // vector vacío
vector<string> nombres(5);    // 5 strings vacíos
vector<float> notas = {7.5, 8.0}; // inicialización con valores
```

💡 Los vectores son **genéricos**: podés usar cualquier tipo (`int`, `string`, `Persona`, etc.).

---

## 📌 9.3 Agregar elementos (`push_back()`)

```
cpp
CopyEdit
vector<int> v;
v.push_back(10);
v.push_back(20);
```

📌 `push_back()` agrega un elemento **al final del vector**.

---

## 📌 9.4 Acceder a elementos

**Usando índice:**

```
cpp
CopyEdit
cout << v[0];           // primer elemento
```

**Usando `.at()` (más seguro):**

```
cpp
CopyEdit
cout << v.at(1);        // segundo elemento
```

⚠️ `.at()` lanza excepción si el índice está fuera de rango.

---

## 📌 9.5 Tamaño del vector

cpp

CopyEdit

```
cout << v.size();    // cantidad de elementos
```

✅ `size()` devuelve el número actual de elementos.

---

## 📌 9.6 Recorrer un vector

**Con índice:**

cpp

CopyEdit

```
for (int i = 0; i < v.size(); i++)  
    cout << v[i] << endl;
```

**Con `auto` (desde C++11):**

cpp

CopyEdit

```
for (auto elem : v)  
    cout << elem << endl;
```

🧠 Se usa comúnmente en bucles tipo `for-each`.

---

## 📌 9.7 Eliminar elementos

**Borrar el último:**

cpp

CopyEdit

```
v.pop_back();
```

**Borrar uno en posición específica:**

cpp

CopyEdit

```
v.erase(v.begin() + 2); // elimina el tercer elemento
```

 `erase()` requiere un **iterador** (`v.begin()` apunta al inicio).

---

## 9.8 Insertar elementos

cpp

CopyEdit

```
v.insert(v.begin() + 1, 99); // inserta 99 en la segunda posición
```

---

## 9.9 Otros métodos útiles

Método	Función
<code>empty()</code>	<code>true</code> si el vector está vacío
<code>clear()</code>	elimina todos los elementos
<code>resize(n)</code>	cambia el tamaño del vector (agrega o recorta)
<code>front()</code> / <code>back()</code>	devuelve el primer / último elemento
<code>assign(n, val)</code>	asigna <code>n</code> veces el valor <code>val</code>

---

## 9.10 Notas importantes

- Los vectores **almacenan datos contiguos en memoria**
  - Son **dinámicos**, crecen automáticamente si hacés `push_back()`
  - Usan **memoria heap** (no stack como los arrays tradicionales)
- 



## PREGUNTAS TIPO PARCIAL – UNIDAD 9

---

❶ ¿Qué función se utiliza para agregar un elemento al final de un **vector**?



- A. `add()`
- B. `insert()`
- C. `append()`
- D. `push_back()`

✓ **Respuesta correcta:** D

**Justificación:** `push_back()` agrega un nuevo elemento al **final del vector**.

---

② ¿Qué valor devuelve `v.size()` en un vector `v`?

- A. El número máximo que puede contener
- B. El número de posiciones disponibles
- C. El número de elementos actualmente almacenados
- D. El tipo de datos del vector

✓ **Respuesta correcta:** C

**Justificación:** `size()` indica la **cantidad de elementos cargados**, no la capacidad total.

---

③ ¿Cuál es la diferencia entre `v[2]` y `v.at(2)`?

- A. `v.at(2)` no permite modificar el valor
- B. `v[2]` solo se usa con strings
- C. `v[2]` lanza excepción si el índice es inválido
- D. `v.at(2)` lanza excepción si el índice está fuera de rango

✓ **Respuesta correcta:** D

**Justificación:** `v.at()` verifica los límites y lanza una **excepción** si se accede a una posición inválida.

---

④ ¿Qué resultado se obtiene al ejecutar este código?

```
cpp
CopyEdit
vector<int> v = {1, 2, 3, 4};
v.erase(v.begin() + 1);
cout << v[1];
```

- A. 2
- B. 3
- C. 1
- D. 4

✓ Respuesta correcta: B

Justificación: Se elimina el segundo elemento (2), por lo que `v[1]` pasa a ser 3.

---

⑤ ¿Qué hace el método `clear()` en un vector?

- A. Elimina solo el primer elemento
- B. Elimina todos los elementos del vector
- C. Reinicia los valores a cero
- D. Reordena el contenido

✓ Respuesta correcta: B

Justificación: `clear()` borra todos los elementos del vector, dejándolo vacío.

## UNIDAD 10: "ENTRADA Y SALIDA EN ARCHIVOS"

📌 ¿Qué es un *stream*?

- Un **stream** (flujo) es una secuencia de datos que **se lee o se escribe**.
- En C++, los **flujos estándar** se gestionan con la librería `<iostream>`:
  - `cin`: entrada estándar (teclado)
  - `cout`: salida estándar (pantalla)

📌 **Mostrar elementos en pantalla**

Se utiliza el operador de inserción (`<<`) con `cout`:

```
cpp
CopyEdit
int x = 5;
cout << "El valor es: " << x << endl;
```

---

📌 **Mostrar elementos de un `vector`**

### Ejemplo directo:

```
cpp
CopyEdit
vector<int> v = {1, 2, 3};
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}
```

### Con bucle **for-each** (C++11+):

```
cpp
CopyEdit
for (int x : v) {
    cout << x << " ";
}
```

✅ Esta forma es más legible y evita errores de índice.

---

## Funciones para mostrar vectores

Se puede **encapsular la lógica** en una función para reutilizarla:

```
cpp
CopyEdit
void mostrarVector(const vector<int>& v) {
    for (int x : v) {
        cout << x << " ";
    }
    cout << endl;
}
```

### Claves:

- `const vector<int>&`: se pasa por **referencia constante**, para evitar copias innecesarias y proteger el contenido original.
  - `cout << x << " "`: muestra los elementos separados por espacios.
- 

## Plantillas (templates) para mostrar cualquier tipo de vector

Cuando se quiere hacer la función más **genérica** (que funcione con cualquier tipo de dato), se usa `template`:

cpp

CopyEdit

```
template <typename T>
void mostrar(const vector<T>& v) {
    for (const T& elem : v) {
        cout << elem << " ";
    }
    cout << endl;
}
```

✅ Esto permite reutilizar la misma función para `vector<int>`, `vector<string>`, `vector<MiClase>`, etc.

---

## Mostrar colecciones personalizadas

Si estás trabajando con una clase (por ejemplo, `Persona`), y querés mostrarla con `cout`, podés **sobrecargar el operador <<**:

cpp

CopyEdit

```
class Persona {
    string nombre;
public:
    Persona(string n) : nombre(n) {}
    string getNombre() const { return nombre; }
};

ostream& operator<<(ostream& os, const Persona& p) {
    os << p.getNombre();
    return os;
}
```

✅ Ahora podés hacer `cout << miPersona;` directamente.

---

## Buenas prácticas

- Usar `const&` para evitar copias innecesarias de vectores grandes
  - Separar la lógica de impresión en funciones auxiliares
  - Usar `template` para evitar duplicar funciones por tipo de dato
  - Utilizar `endl` solo cuando sea necesario (consume más rendimiento que `"\n"`)
- 



## PREGUNTAS TIPO PARCIAL – UNIDAD 10

---

❶ ¿Cuál es la principal ventaja de usar `const vector<T>&` como parámetro en una función?

- A. Permite modificar el vector original
- B. Copia el vector para mayor seguridad
- C. Evita copias innecesarias y garantiza que no se modifica el vector
- D. Solo funciona con vectores de tipo `int`

✅ Respuesta correcta: C

**Justificación:** Al usar `const vector<T>&`, se pasa una **referencia constante**, lo que **evita copias de memoria innecesarias** (eficiencia) y **protege el contenido** (no se puede modificar dentro de la función).

---

❷ ¿Qué operador se utiliza para mostrar información en pantalla con `cout`?

- A. `>>`
- B. `<<`
- C. `==`
- D. `->`

✅ Respuesta correcta: B

**Justificación:** El operador `<<` es el operador de **inserción en flujo de salida**, y se utiliza para imprimir en pantalla con `cout`.

---

❸ ¿Qué ventaja ofrece el uso de un `template` en una función de impresión?

- A. Permite imprimir solo tipos primitivos
- B. Permite imprimir vectores sin conocer su tipo exacto

- C. Obliga a usar vectores de enteros
- D. Evita tener que usar `cout`

✓ **Respuesta correcta: B**

**Justificación:** Al declarar una función como `template`, se vuelve **genérica**, lo que permite **trabajar con cualquier tipo de dato**, no solo con `int`.

---

④ ¿Qué efecto tiene la sobrecarga del operador `<<` en una clase?

- A. Permite sumar objetos de esa clase
- B. Permite usar el operador de salida `cout` con objetos de la clase
- C. Permite comparar objetos
- D. Crea una nueva función `main`

✓ **Respuesta correcta: B**

**Justificación:** Al sobrecargar `<<`, se le enseña al compilador cómo debe mostrarse un objeto de esa clase al usarlo con `cout`.

---

⑤ ¿Qué diferencia hay entre `endl` y `"\n"` en C++?

- A. `endl` no existe en C++
- B. `"\n"` hace una pausa, `endl` no
- C. `endl` además de saltar de línea, fuerza el vaciado del búfer de salida
- D. Son completamente equivalentes en todo contexto

✓ **Respuesta correcta: C**

**Justificación:** Ambos provocan un salto de línea, pero `endl` **fuerza el flush** del flujo de salida, lo que puede tener impacto en el rendimiento.

## **UNIDAD 11: "STANDARD TEMPLATE LIBRARY"**

### **Objetivo**

Introducir el uso de la **librería estándar de plantillas de C++ (STL)**, que proporciona **contenedores, algoritmos e iteradores** genéricos listos para ser usados. Esta unidad busca mostrar su utilidad en la programación orientada a objetos, donde la reutilización de código es fundamental.

---

# CONTENIDO DETALLADO

---

## ¿Qué es la STL?

- Es una **colección de clases y funciones genéricas** que implementan estructuras de datos y algoritmos comunes.
  - Permite escribir código más eficiente, reutilizable y menos propenso a errores.
- 

## Principales componentes de la STL

1. **Contenedores**: estructuras que almacenan objetos.
  2. **Iteradores**: permiten recorrer los elementos de los contenedores.
  3. **Algoritmos**: funciones que operan sobre contenedores (ordenar, buscar, contar, etc.).
- 

## Contenedores más usados

Tipo	Descripción
<code>vector</code>	Arreglo dinámico de tamaño variable.
<code>list</code>	Lista doblemente enlazada.
<code>deque</code>	Cola de doble entrada.
<code>stack</code>	Pila (LIFO).
<code>queue</code>	Cola (FIFO).
<code>map</code>	Diccionario clave-valor ordenado.
<code>set</code>	Conjunto de elementos únicos.

---

## Uso básico de `vector`

cpp

CopyEdit

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

vector<int> numeros;
numeros.push_back(10);
numeros.push_back(20);
```

`push_back()` agrega un elemento al final.

Se puede acceder con `numeros[i]`.

---

## Iteradores

Los **iteradores** actúan como punteros para recorrer contenedores.

cpp

CopyEdit

```
vector<int>::iterator it;
for (it = numeros.begin(); it != numeros.end(); ++it) {
    cout << *it << " ";
}
```

- `begin()` → devuelve iterador al primer elemento
- `end()` → devuelve iterador al final (uno después del último)

También se puede usar el **bucle for-each**:

cpp

CopyEdit

```
for (int x : numeros) {
    cout << x << " ";
}
```

---

## Algoritmos de STL

Hay decenas de algoritmos listos para usar en `<algorithm>`:

cpp



CopyEdit

```
#include <algorithm>
sort(numeros.begin(), numeros.end());
```

Algunos comunes:

- `sort()` → ordena elementos
  - `find()` → busca un valor
  - `count()` → cuenta apariciones
  - `reverse()` → revierte el orden
  - `max_element()` / `min_element()` → devuelve el mayor/menor
- 

## Ventajas de la STL

- Código más limpio y legible
  - Mejora en eficiencia
  - Evita errores al no reinventar estructuras comunes
  - Fácil de combinar con programación orientada a objetos
- 

## Buenas prácticas

- Usar el **tipo `auto`** para declarar iteradores si se permite (C++11+)
  - Aprovechar los algoritmos antes de escribir funciones propias
  - Tener cuidado con la **complejidad algorítmica** (algunos algoritmos pueden ser costosos)
- 



## PREGUNTAS TIPO PARCIAL – UNIDAD 11

---

❶ ¿Qué es la STL en C++?

- A. Una librería de gráficos
- B. Una librería de funciones matemáticas
- C. Una colección de clases genéricas para estructuras de datos y algoritmos
- D. Una herramienta para depurar código

✅ **Respuesta correcta: C**

**Justificación:** STL es la *Standard Template Library*, una colección de clases y funciones para trabajar con estructuras de datos y algoritmos de manera genérica.

---

❷ ¿Cuál de las siguientes NO es un contenedor STL?

- A. `vector`
- B. `map`
- C. `arraylist`
- D. `set`

✅ **Respuesta correcta: C**

**Justificación:** `arraylist` es un tipo de estructura en Java, **no** existe como contenedor en la STL de C++.

---

❸ ¿Qué hace la función `sort()` aplicada a un `vector<int>`?

- A. Invierte el orden de los elementos
- B. Ordena los elementos de menor a mayor
- C. Elimina duplicados
- D. Devuelve el promedio de los elementos

✅ **Respuesta correcta: B**

**Justificación:** `sort()` ordena los elementos entre los iteradores `begin()` y `end()` en orden creciente por defecto.

---

❹ ¿Cuál es el rol de los iteradores en la STL?

- A. Almacenan datos
- B. Ejecutan algoritmos
- C. Recorren contenedores
- D. Compilan funciones

✓ Respuesta correcta: C

**Justificación:** Los iteradores funcionan como punteros para recorrer los elementos de los contenedores STL.

---

⑤ ¿Qué función STL se usa para contar cuántas veces aparece un valor?

- A. `find()`
- B. `count()`
- C. `match()`
- D. `index()`

✓ Respuesta correcta: B

**Justificación:** `count(begin, end, valor)` devuelve la cantidad de veces que aparece un valor determinado en una colección.

## UNIDAD 12: “ALGUNAS ESTRUCTURAS ÚTILES”

### 📌 `pair<T1, T2>`

- Estructura que agrupa **dos valores de distinto (o igual) tipo**.
- Muy útil para representar **relaciones entre dos datos** (como clave–valor, coordenadas, etc.).

```
cpp
CopyEdit
#include <utility>
pair<int, string> dato(1, "Lucas");

cout << dato.first << " " << dato.second;
```

- `first`: primer elemento del par
- `second`: segundo elemento

✓ Usado frecuentemente en `map`, algoritmos, retorno de múltiples valores.

---

## **tuple<T1, T2, T3, ...>**

- Similar a **pair**, pero puede contener **más de dos valores**.

```
cpp
CopyEdit
#include <tuple>
tuple<int, string, double> persona(1, "Lucas", 9.5);
```

Para acceder a los elementos se usa **get<>**:

```
cpp
CopyEdit
cout << get<0>(persona) << " " << get<1>(persona);
```

- ✓ Muy útil cuando se necesita devolver **múltiples valores heterogéneos** desde una función.
- 

## **map<Key, Value>**

- Contenedor que almacena pares **clave-valor**, con claves únicas y ordenadas.

```
cpp
CopyEdit
#include <map>
map<string, int> edades;

edades["Lucas"] = 25;
edades["Ana"] = 30;

cout << edades["Ana"]; // Muestra 30
```

- Las claves se ordenan automáticamente (por defecto usando **<**).
  - Si se accede a una clave que no existe, **se crea automáticamente con el valor por defecto del tipo**.
-

## **set<T>**

- Contenedor que **almacena elementos únicos y ordenados**.

```
cpp
CopyEdit
#include <set>
set<int> numeros = {3, 1, 4, 1, 5};

for (int n : numeros) {
    cout << n << " ";
}
// Salida: 1 3 4 5
```

- ✓ No permite duplicados. Se usa para representar **conjuntos matemáticos**.
- 

## **Otras estructuras útiles en STL (nombradas)**

- **stack**: pila (LIFO)
- **queue**: cola (FIFO)
- **priority\_queue**: cola con prioridad
- **unordered\_map**: como **map** pero sin orden (más eficiente en ciertas operaciones)

Estas estructuras están pensadas para cubrir necesidades frecuentes sin tener que implementarlas desde cero.

---

## **Comparación breve**

Estructura	Propósito principal
<b>pair</b>	Relación simple de dos valores
<b>tuple</b>	Múltiples valores heterogéneos
<b>map</b>	Asociación clave–valor
<b>set</b>	Conjunto de elementos únicos ordenados

stack/queue Estructuras de acceso por reglas específicas

---



## PREGUNTAS TIPO PARCIAL – UNIDAD 12

---

❶ ¿Cuál es la diferencia entre **pair** y **tuple**?

- A. **pair** permite más elementos que **tuple**
- B. **tuple** permite más de dos elementos, **pair** solo dos
- C. Ambas son iguales
- D. **tuple** es más rápida pero menos segura

✅ Respuesta correcta: B

**Justificación:** **pair** solo puede almacenar dos elementos, mientras que **tuple** puede almacenar tres o más, todos de tipos distintos.

---

❷ ¿Qué hace el contenedor **set** en C++?

- A. Permite valores duplicados
- B. Ordena elementos y permite duplicados
- C. Almacena elementos únicos en orden automático
- D. Almacena pares clave-valor

✅ Respuesta correcta: C

**Justificación:** **set** almacena elementos únicos y los mantiene ordenados automáticamente.

---

❸ ¿Cuál es el comportamiento de **map** si se accede a una clave inexistente?

- A. Lanza un error
- B. No hace nada
- C. Crea la clave con valor por defecto
- D. Devuelve **null**

✅ Respuesta correcta: C

**Justificación:** **map** crea automáticamente una nueva entrada con el valor por defecto del tipo de dato si se accede a una clave que no existía.

---

④ ¿Para qué sirve `get<N>(tupla)`?

- A. Elimina el elemento N
- B. Crea una tupla
- C. Modifica un valor
- D. Accede al elemento N de la tupla

✓ Respuesta correcta: D

Justificación: `get<N>()` permite acceder directamente al elemento N de una `tuple`.

---

⑤ ¿Cuál de estas estructuras sería más adecuada para representar coordenadas (x, y) en un punto?

- A. `tuple<int, int, int>`
- B. `map<string, int>`
- C. `pair<int, int>`
- D. `set<int>`

✓ Respuesta correcta: C

Justificación: `pair<int, int>` es la forma más simple y clara de representar un par de valores relacionados, como coordenadas.