1. *Compilación y Vinculación de Librerías*

En esta sección se estudian los **lenguajes compilados e interpretados** y cómo gestionan la **instalación, compilación y vinculación de librerías**. Se trabajará con **C, Rust, Java y Python**, una vez completados los pasos de instalación del anexo.

1.1. *Fundamentos del proceso de compilación y ejecución*

El proceso de **compilación** transforma el código fuente en un **programa ejecutable** mediante varias etapas sucesivas:

- **Preprocesado** — Expansión de macros y directivas (`#include`, `#define` en C).
- **Compilación** — Traducción del código fuente a ensamblador o código intermedio.
- **Ensamblado** — Generación de archivos objeto (`.o`, `.obj`) con código máquina parcial.
- **Vinculación (linking)** — Combinación de archivos objeto y librerías para formar el ejecutable final.

En un ejecutable final: Si se usaron **librerías estáticas**, su código está **incluido** dentro del binario. Si se usaron **dinámicas**, el binario solo guarda **enlaces simbólicos** a ellas, y el sistema operativo las carga al ejecutar el programa.

1.2. *Lenguajes compilados e interpretados*

Cada entorno combina la compilación y el linking de forma diferente: en C o Rust, las librerías deben estar compiladas y disponibles; en Java o Python, el enlace ocurre en ejecución, al cargar el bytecode o el módulo.

- **C/C++**: Compilan cada archivo fuente (`.c`, `.cpp`) en objetos (`.o`, `.obj`), que luego se **vinculan con librerías** para formar el ejecutable. Las librerías pueden ser: Estáticas **(`.a`, `.lib`): su código se integra directamente en el ejecutable. Dinámicas** (`.so`, `.dll`, `.dylib`): se cargan en tiempo de ejecución. Herramientas como `make` o `cmake` automatizan el proceso, recompilando solo lo necesario y resolviendo dependencias externas.
- **Rust**: Usa el compilador `rustc` y el gestor de paquetes `cargo`, que combinan **compilación, vinculación y manejo de dependencias**. Cada proyecto define sus librerías y dependencias en `Cargo.toml`. Rust genera archivos objeto y ejecutables igual que C, pero añade una fase de **verificación de seguridad de memoria** antes del enlace. Las librerías pueden compilarse como **crates** estáticos (`.rlib`) o dinámicos (`.so`, `.dll`), y `cargo` gestiona automáticamente la vinculación.

```
Código fuente → (compilación) → Código objeto → (vinculación) → Ejecutable
                      ↑                                  ↓
            Librerías estáticas                Librerías dinámicas
```

- **Java**: Compila los archivos `.java` a **bytecode** (`.class`), que ejecuta la **JVM (Java Virtual Machine)**. Las librerías se distribuyen como **archivos JAR** (`.jar`), que son paquetes ZIP con clases compiladas y metadatos. El programa principal puede usar librerías externas añadiéndolas al **classpath**. Los sistemas de construcción (`Maven`, `Gradle`) descargan y gestionan automáticamente dependencias desde repositorios como **Maven Central**.

```
Código fuente (.java) → (compilación) → Bytecode (.class) → (ejecución) → JVM
                                  ↑
                        Librerías externas (.jar)
```

- **Python**: Es un lenguaje **interpretado**, pero muchas de sus librerías contienen **extensiones compiladas** en C o C++. Los paquetes pueden incluir:

  -Archivos `.py` (código interpretado).

  -Archivos **bytecode** (`.pyc`).

  -Archivos **binarios compilados** (`.so` en Linux, `.pyd` en Windows).

  Las librerías se suelen distribuir en formato **wheel** (`.whl`), un paquete precompilado que ya contiene los binarios listos para cada plataforma. Así, al instalar con `pip install`, normalmente no se compila nada localmente: se descargan los binarios adecuados desde **PyPI**.

```
  Códigofuente(.py) → (compilación interna)Bytecode(.pyc) → (ejecución)Intérprete Python
                                                ↑
                                  Extensiones compiladas (.so / .pyd)
```

Lenguaje	Herramientas de compilación	Gestión de librerías
C	`gcc`, `make`, `cmake`	`pkg-config`, rutas del sistema
Rust	`cargo` (usa `rustc` internamente)	`Cargo.toml`, crates.io
Java	`javac`, `jar`	`Maven`, `Gradle`
Python	Intérprete `python`	`pip`, `requirements.txt`, `poetry`

1.3. *Vinculación y uso de librerías*

Las **librerías** son colecciones de código precompilado que extienden la funcionalidad de un programa sin necesidad de reescribirla.

1.3.1. *Tipos principales*

Librerías estáticas (`.a`, `.lib`): Se integran directamente en el ejecutable durante la compilación. El resultado final contiene todo el código necesario y no depende de archivos externos.

Librerías dinámicas (`.so`, `.dll`, `.dylib`): Se cargan en tiempo de ejecución. El ejecutable solo guarda una **referencia simbólica** a ellas, por lo que deben estar presentes en el sistema al ejecutar el programa.

Ejemplo: En C, el comando `gcc main.c -lm` vincula el programa con la librería matemática (`libm.so` en Linux). En Rust, Cargo gestiona automáticamente las dependencias desde `Cargo.toml`. En Java, las librerías se empaquetan como `.jar`. En Python, se instalan con `pip` y pueden incluir extensiones compiladas (`.so`, `.pyd`).

1.4. 🔧 *Ejercicio Práctico n.º 1*

Compilar un programa sencillo en **C**, **Rust**, **Java** y **Python**, instalando e importando **una librería externa** en cada caso.

Comprobar:

- Cómo se vincula la librería (estática o dinámica).
- Dónde se encuentra tras compilar o ejecutar el programa.
- Si el binario resultante depende de librerías externas.

2. *Memoria*

2.1. *Scope de una variable*

El scope define dónde una variable es accesible.

- Global: Disponible en todo el programa
- Función: Accesible solo dentro de una función
- Bloque: Accesible dentro de {} o similar
- Clase: Atributos accesibles dentro del objeto o clase

Reglas de scope por lenguaje:

- C: Global, función, bloque
- Java: Clase, método, bloque
- Python: LEGB (Local, Enclosing, Global, Built-in)
- Rust: Sistema de ownership determina visibilidad y duración

2.2. *Tipos de memoria*

Stack: Memoria temporal para variables locales y llamadas a funciones. Se libera automáticamente al salir del scope.

Heap: Espacio dinámico para objetos o estructuras creadas en tiempo de ejecución.

Segmento estático: Contiene variables globales o estáticas.

Segmento de código: Contiene las instrucciones ejecutables del programa.

2.3. *Gestión de memoria*

La gestión de memoria determina cómo un programa reserva, usa y libera espacio durante su ejecución. Existen tres modelos principales: gestión manual, automática mediante garbage collector, y basada en ownership.

2.3.1. *Ciclo de vida de la memoria*

Reserva: El sistema asigna espacio (en stack o heap).

Uso: El programa accede y modifica los datos.

Liberación: La memoria se devuelve al sistema (manual o automática).

2.3.2. *Estrategias según el lenguaje*

- Gestión manual (C, C++, Rust):

El programador **gestiona la memoria directamente** con funciones o reglas de propiedad.

En C y C++, se usa malloc/free o new/delete.

En Rust, el compilador garantiza la liberación automática gracias a su sistema de ownership y borrowing, sin necesidad de recolector de basura.

- Recolección de basura (Java, Python):

La memoria se libera **automáticamente**.

Emplean técnicas como mark-and-sweep o GC generacional.

Evitan fugas de memoria, aunque pueden introducir pausas o sobrecarga.

Modelos mixtos:

Algunos lenguajes combinan conteo de referencias con detección de ciclos (por ejemplo, Python).

2.3.3. *Práctica por lenguaje*

C (gestión manual): El programador debe reservar y liberar memoria manualmente. Por ejemplo, al usar malloc se asigna espacio en el heap y se debe liberar con free. Si se omite la liberación, se produce una fuga de memoria. Las variables locales (en stack) se crean y destruyen automáticamente al entrar y salir de una función.

```c
char *mensaje = malloc(20);
strcpy(mensaje, "hola");
printf("%s\n", mensaje);
free(mensaje); // libera la memoria
```

Buena práctica : Emparejar siempre cada malloc con un free.

Java (recolección de basura): Las variables locales se almacenan en el stack, mientras que los objetos (new) se guardan en el heap. El Garbage Collector detecta qué objetos ya no tienen referencias y los libera. Aunque el programador no gestiona directamente la memoria, puede influir en el rendimiento evitando crear objetos innecesarios.

```java
String saludo = new String("hola");
System.out.println(saludo);
```

Buena práctica: preferir variables locales o reutilizar objetos para reducir carga del GC.

Python (conteo de referencias + GC): Todo en Python es un objeto en el heap. Cada objeto lleva un contador de referencias, que se incrementa o decrementa según el uso. Cuando el contador llega a cero, la memoria se libera. Además, el recolector de basura detecta ciclos de referencia.

```python
a = [1, 2, 3]
b = a        # ambas variables apuntan a la misma lista
a = None     # se reduce el contador de referencias
```

Buena práctica: eliminar referencias circulares (por ejemplo, con weakref) y evitar mantener objetos vivos innecesariamente.

Rust (ownership): Rust no usa un recolector de basura. En su lugar, aplica un sistema de ownership (propiedad), que define quién es responsable de liberar la memoria. **Cada valor en Rust tiene un único owner**, es decir, una entidad que controla su ciclo de vida y libera su memoria automáticamente cuando sale del scope.

¿Quién puede ser un owner?

- Una variable local que almacena un valor.
- Una función que recibe el valor como argumento (por valor, no por referencia).
- Una estructura (struct) que contiene el valor como campo.
- Una colección (Vec, HashMap, etc.) que guarda el valor como elemento.

Cuando el owner desaparece, Rust libera automáticamente la memoria de ese valor. No hace falta usar free() ni un recolector de basura.

```rust
let s1 = String::from("hola");
let s2 = s1; // la propiedad pasa de s1 a s2
println!("{}", s1); // error: s1 ya no es válido
```

Para acceder a un valor sin transferir la propiedad, Rust permite prestar datos mediante referencias (borrowing):

&T → préstamo inmutable (solo lectura)

&mut T → préstamo mutable (lectura y escritura exclusivas)

```rust
let s1 = String::from("hola");
let s2 = &s1; // préstamo inmutable
println!("{}", s1); // válido, sigue siendo owner
println!("{}", s2); // también válido
```

Buena práctica: comprender las reglas de ownership y usar borrowing (&T, &mut T) para acceder sin transferir propiedad.

2.4. *Rendimiento y seguridad de memoria*

C/C++: Máximo control y rendimiento, pero riesgo de errores graves (fugas, punteros colgantes).

Java/Python: Más seguros y sencillos, aunque con pausas del recolector y menor eficiencia.

Rust: Equilibra rendimiento y seguridad, evitando los problemas clásicos de C sin necesidad de GC.

🔧 Ejercicio Práctico n.º 2 — Realizar por separado Comparar el comportamiento de memoria y visibilidad de variables entre C, Java, Python y Rust. Analizar cuándo se crean, acceden y destruyen variables, y qué mecanismos de control de memoria emplea cada lenguaje.

3. *Lenguaje orientado a objetos*

La orientación a objetos no es un modelo de cómputo, sino una forma de organizar los programas.

Permite:

- **Modularizar y encapsular el código**
- **Reutilizar componentes mediante herencia o composición**
- **Abstraer detalles de implementación**
- **Adaptarse bien a proyectos grandes y colaborativos**

Conceptos clave:

- **Encapsulación**: Oculta la implementación interna
- **Herencia**: Permite definir nuevas clases a partir de otras
- **Polimorfismo**: Permite tratar objetos diferentes con la misma interfaz
- **Abstracción**: Representar entidades relevantes en el código

Implementación en distintos lenguajes:

- **Java**: Orientación a objetos pura (todo menos primitivos como int o double es objeto)
- **Python**: Muy flexible, todo es objeto
- **C++**: Híbrido (orientado a objetos + procedural)

- **Rust**: Usa `traits` y `structs` para lograr encapsulación sin herencia

🔧 **Ejercicio Práctico n.º 3 — Realizar por separado**

Implementar una jerarquía simple de clases o estructuras en al menos dos lenguajes (Java y Python, o C++ y Rust), usando encapsulación, herencia/composición y polimorfismo.

—

4. *Bibliografía*

- Aho, Alfred V. et al. **Compilers: Principles, Techniques, and Tools**. Addison-Wesley, 2006. Fundamentos de compilación y ejecución.
- Stroustrup, Bjarne. **The C++ Programming Language**. Addison-Wesley, 2013. Programación, gestión de memoria y POO.
- Matsakis, Nicholas D., y Felix S. Klock II. **The Rust Programming Language**. No Starch Press, 2019. Seguridad y manejo de memoria en Rust.
- Sebesta, Robert W. **Concepts of Programming Languages**. Pearson, 2015. Paradigmas, sintaxis y semántica.

4.1. *Anexo: Instalación de lenguajes y librerías*

Describimos como instalar en Windows, macOS y Ubuntu, en caso de que tu distro de linux sea distinta te supondremos capaz de encontrar la forma de instalar el lenguaje.

4.2. *C*

C y C++:

- Para Windows: instalar **MinGW** o **Visual Studio** (incluye compilador).
- Para macOS: instalar Xcode Command Line Tools (`xcode-select --install`).
- Para Linux(Ubuntu): instalar `build-essential` (`sudo apt install build-essential`).

Compilar y ejecutar:

```
gcc programa.c -o programa   # C
g++ programa.cpp -o programa # C++
./programa
```

Librerías externas:

Se suelen descargar y compilar manualmente o usar package managers **como** vcpkg **o** Conan**.** Ejemplo con vcpkg:

```
vcpkg install nombre_libreria
```

4.3. *Java*

- **Windows:**
  - Descargar el instalador del JDK desde https://jdk.java.net y seguir el asistente.
  - Verificar instalación:

    ```
    java -version
    javac -version
    ```

- Compilar y ejecutar:

```
javac Programa.java
java Programa
```

- **macOS:**
  - Instalar JDK desde jdk.java.net o usando Homebrew:

    ```
    brew install openjdk
    ```

  - Verificar instalación:

    ```
    java -version
    javac -version
    ```

  - Compilar y ejecutar igual que Windows.

- **Linux (Ubuntu/Debian):**

  ```
  sudo apt update
  sudo apt install default-jdk
  java -version
  javac -version
  ```

  - Compilar y ejecutar igual que Windows.

Instalar librerías:

- Usar **Maven** o **Gradle** para gestionar dependencias.
- Ejemplo con Maven (agregar al `pom.xml`):

```
<dependency>
    <groupId>org.example</groupId>
    <artifactId>nombre-libreria</artifactId>
    <version>1.0.0</version>
</dependency>
```

4.4. *Python*

4.4.1. *Windows*

- Descargar desde https://www.python.org/downloads
- Marcar "Add Python to PATH" al instalar
- Verificar: `python --version` y `pip --version`

4.4.2. *macOS*

- Descargar desde https://www.python.org/downloads o con Homebrew: `brew install python`
- Verificar: `python3 --version` y `pip3 --version`

4.4.3. *Linux*

```
sudo apt install python3 python3-pip  # Ubuntu/Debian
```

Instalar librerías:

```
pip install nombre_libreria  # Windows
pip3 install nombre_libreria # macOS/Linux
```

Entornos virtuales (recomendado):

```
python -m venv mi_entorno
# Activar:
mi_entorno\Scripts\activate     # Windows
source mi_entorno/bin/activate  # macOS/Linux
pip install nombre_libreria
```

4.5. *Rust*

4.5.1. *Windows*

- Descargar rustup desde https://rustup.rs
- Ejecutar `rustup-init.exe`
- Reiniciar terminal

4.5.2. *macOS/Linux*

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source "$HOME/.cargo/env"
```

Verificar instalación:

```
rustc --version
cargo --version
```

Compilar y ejecutar:

```
cargo new mi_proyecto
cd mi_proyecto
cargo run
```

Instalar librerías (crates):

- Se gestionan en `Cargo.toml`:

```
[dependencies]
rand = "0.9"
```

- Instalar automáticamente con `cargo build` o `cargo run`