Universidad Autónoma de Yucatán

Facultad de Matemáticas

Ingeniería de Software

Teoría de Lenguajes de Programación

Proyecto Final - Segunda Entrega (Rust)

Autores:
Canto Vázquez Esteban
Ceballos Medina José Manuel
Puch Uribe Ángel Leandro
Ramírez Ceciliano Mauricio Emiliano

Profesor: Luis Fernando Curi Quintal

Fecha de entrega: 27/05/2025

Implementación del lenguaje de programación (Rust)

El lenguaje de programación empleado en este proyecto es Rust, un lenguaje de propósito general orientado al desarrollo de sistemas, diseñado para proporcionar una alternativa segura y moderna a lenguajes como C y C++. Su principal enfoque es garantizar la seguridad de memoria sin necesidad de un recolector de basura, además de ofrecer un alto rendimiento y soporte para concurrencia segura.

Rust es un lenguaje compilado, lo que significa que el código fuente debe ser transformado en un binario ejecutable antes de poder ejecutarse. Esta compilación se realiza mediante el compilador oficial rustc, que forma parte de la herramienta de gestión de proyectos y dependencias llamada cargo. El compilador no solo se encarga de generar código optimizado, sino que también verifica reglas estrictas de seguridad en tiempo de compilación, como el sistema de ownership y borrowing, que impide errores comunes como el uso de memoria después de haber sido liberada o condiciones de carrera en ejecuciones concurrentes.

En este proyecto se utilizó rustc a través del comando cargo run, lo que permite compilar y ejecutar el programa en un solo paso. El proceso de compilación convierte los archivos .rs (código fuente en Rust) en binarios que pueden ejecutarse directamente sobre el sistema operativo, sin depender de una máquina virtual ni de un intérprete. Esta característica es fundamental para el tipo de aplicación desarrollada, ya que se buscó un comportamiento eficiente, seguro y con acceso directo al sistema de archivos para la lectura y escritura de información persistente.

Además, al ser un lenguaje compilado y fuertemente tipado, Rust permite detectar la mayoría de los errores de lógica y tipo en tiempo de compilación, lo cual reduce significativamente los errores en tiempo de ejecución y facilita el mantenimiento del código a largo plazo.

Ambiente de programación

El desarrollo de esta aplicación en Rust se realizó en el entorno de trabajo Visual Studio Code (VS Code), un editor de código fuente ligero, multiplataforma y altamente extensible, ampliamente adoptado por la comunidad de desarrolladores. VS Code fue elegido por su integración fluida con el ecosistema de Rust, su soporte para depuración, resaltado de sintaxis, ejecución de tareas y administración de proyectos.

Para facilitar el desarrollo en Rust dentro de VS Code, se instaló la extensión oficial rust-analyzer, que proporciona funcionalidades esenciales como:

- Autocompletado inteligente.
- Detección temprana de errores y advertencias.
- Resaltado de sintaxis y navegación entre archivos y símbolos.
- Soporte para refactorizaciones y sugerencias contextuales.

Además, el entorno se complementó con el uso de Cargo, el gestor de paquetes y compilación de Rust. Cargo se encarga de:

- Crear la estructura base del proyecto.
- Compilar el código mediante el compilador rustc.
- Ejecutar el binario generado.
- Instalar y administrar dependencias externas como serde y serde_json, utilizadas en este proyecto para serialización de datos.

El sistema operativo utilizado durante el desarrollo fue Windows 11, aunque el entorno es totalmente portátil y multiplataforma. Esto significa que el mismo proyecto podría compilarse y ejecutarse en Linux o macOS sin requerir cambios en el código, gracias a la portabilidad del lenguaje Rust y sus herramientas.

Este ambiente permitió una experiencia de desarrollo ágil, confiable y organizada, facilitando tanto la codificación como las pruebas e iteraciones sobre la aplicación.

Organización de la aplicación

La aplicación fue organizada de manera estructurada, utilizando los principios fundamentales del lenguaje Rust para mantener un código modular, legible y mantenible. Todo el programa está contenido en un único archivo fuente (main.rs), siguiendo una organización lógica basada en funciones, estructuras de datos y control de flujo por menús.

Menú principal y control de flujo

El programa implementa un **menú interactivo en consola**, el cual guía al usuario a través de las diferentes funcionalidades disponibles:

1. Agregar un contacto.
2. Consultar un contacto por nombre.
3. Listar los contactos con cumpleaños en un mes específico.

4. Listar todos los contactos ordenados alfabéticamente.
5. Eliminar un contacto existente.
6. Salir del programa.

Este menú se ejecuta dentro de un bucle principal (loop**)**, que permanece activo hasta que el usuario elige salir. Las opciones del menú son gestionadas con una instrucción match, que distribuye el flujo de ejecución a la función correspondiente, según la elección del usuario.

Funciones y procedimientos

Cada funcionalidad del programa fue implementada como una función separada, lo que permite encapsular la lógica y facilitar la reutilización y pruebas. Por ejemplo:

- agregar_contacto() encapsula todo el proceso de recolección, validación y almacenamiento de un nuevo contacto.
- buscar_contacto() recibe un nombre y muestra la información del contacto si existe.
- listar_por_mes() filtra y muestra solo los contactos que cumplen años en el mes especificado.
- eliminar_contacto() permite eliminar un contacto por nombre, con actualización automática del archivo de almacenamiento.
- Otras funciones auxiliares como validar_dia() o pedir_numero() facilitan la entrada y validación de datos.

Modularidad y claridad

Aunque toda la aplicación se encuentra en un único archivo fuente para facilitar su entrega y ejecución en entornos académicos, el código ha sido organizado en bloques funcionales claramente diferenciados mediante comentarios y separaciones visuales. En proyectos más grandes, este diseño puede fácilmente escalarse dividiendo el código en módulos (mod) y archivos independientes.

Este enfoque modular no solo mejora la mantenibilidad, sino que también favorece la detección de errores, la depuración y futuras ampliaciones del sistema, como podría ser la exportación a formatos CSV o la integración con una interfaz gráfica.

Estructuras de datos utilizadas

La aplicación hace uso de estructuras de datos fundamentales del lenguaje Rust para representar, almacenar, manipular y persistir la información de los contactos.

Estructura principal: *struct Contacto*

Para representar a cada contacto, se definió una estructura personalizada llamada *Contacto*, declarada mediante la palabra clave *struct*. Esta estructura contiene los siguientes campos:

```
struct Contacto {
    nombre: String,
    cumple: String,
    telefono: String,
    correo: String,
}
```

Cada campo es de tipo *String* y representa respectivamente el nombre, la fecha de cumpleaños (en formato dd/mm), el número telefónico y el correo electrónico del contacto.

Además, se utilizó el atributo *#[derive(Serialize, Deserialize, Debug, Clone)]* sobre la estructura para habilitar la serialización y deserialización de los datos usando las bibliotecas externas *serde* y *serde_json*, lo que permitió leer y escribir los contactos como objetos JSON fácilmente.

Estructura de almacenamiento en memoria: *Vec<Contacto>*

Durante la ejecución del programa, todos los contactos se almacenan en un vector dinámico de tipo *Vec<Contacto>,* una estructura de datos provista por Rust que permite almacenar una lista ordenada de elementos del mismo tipo. Esta colección permite agregar, eliminar, ordenar y recorrer contactos de forma eficiente.

Por ejemplo, al iniciar el programa, los datos se cargan desde el archivo *contactos.json* y se almacenan en un *Vec<Contacto>,* que se modifica en tiempo real a medida que el usuario realiza operaciones. Al salir del programa (o después de agregar/eliminar), este vector se guarda nuevamente en el archivo.

Estructura para ordenamiento: *BTreeMap*

Para listar los contactos en orden alfabético por nombre, se utilizó un *BTreeMap*, una estructura que almacena pares clave-valor ordenados. Al insertar los contactos

en este mapa con el **nombre como clave**, se garantiza que la salida esté ordenada sin necesidad de realizar un paso adicional de ordenamiento.

Almacenamiento persistente: *archivo JSON*

Para mantener la persistencia de los datos entre ejecuciones, se utiliza un archivo llamado *contactos.json*. Este archivo es leído al inicio del programa mediante *serde_json::from_reader()* y sobrescrito cuando hay modificaciones con *serde_json::to_writer_pretty().* Al usar el formato JSON:

- Se facilita la legibilidad del archivo.
- Se asegura compatibilidad con otros lenguajes o herramientas en caso de futura interoperabilidad.

Entrada, procesamiento y salida

- **Entrada**: Datos ingresados por el usuario mediante teclado (*stdin*).
- **Procesamiento**: Validaciones, filtrado, ordenamiento, búsqueda y modificación de la colección *Vec<Contacto>.*
- **Salida**: Presentación de resultados en consola (*println!*) y escritura estructurada de datos en archivo JSON.

Código fuente

```
use serde::{Deserialize, Serialize}; // Para serializar/deserializar datos en formato JSON
use std::fs::File;                   // Para abrir y escribir archivos
use std::io::{self, BufReader};      // Para entrada/salida y lectura eficiente
use std::path::Path;                 // Para verificar si un archivo existe
use std::collections::BTreeMap;      // Para ordenar contactos por nombre


// Estructura que representa a un contacto
#[derive(Serialize, Deserialize, Debug, Clone)]
struct Contacto {
    nombre: String,
```

```
    cumple: String,   // Fecha en formato "dd/mm"
    telefono: String,
    correo: String,
}

// Carga la lista de contactos desde un archivo JSON si existe, o retorna una lista vacía
fn cargar_contactos() -> Vec<Contacto> {
    if !Path::new("contactos.json").exists() {
        return vec![]; // Si el archivo no existe, se retorna una lista vacía
    }
    let file = File::open("contactos.json").unwrap();
    let reader = BufReader::new(file);
   serde_json::from_reader(reader).unwrap_or_else(|_| vec![]) // Se deserializa el contenido
}

// Guarda la lista de contactos en un archivo JSON con formato legible
fn guardar_contactos(contactos: &[Contacto]) {
    let file = File::create("contactos.json").unwrap();
    serde_json::to_writer_pretty(file, &contactos).unwrap();
}

// Solicita al usuario un número y valida que sea entero
fn pedir_numero(mensaje: &str) -> u32 {
    loop {
        let mut entrada = String::new();
```

```
        println!("{}", mensaje);
        io::stdin().read_line(&mut entrada).unwrap();
        if let Ok(num) = entrada.trim().parse::<u32>() {
            return num;
        } else {
            println!("Entrada inválida. Intenta de nuevo.");
        }
    }
}

// Verifica si el día ingresado es válido para el mes correspondiente
fn validar_dia(dia: u32, mes: u32) -> bool {
    match mes {
        1 | 3 | 5 | 7 | 8 | 10 | 12 => dia >= 1 && dia <= 31,
        4 | 6 | 9 | 11 => dia >= 1 && dia <= 30,
        2 => dia >= 1 && dia <= 29, // Aceptamos hasta 29 días en febrero
        _ => false,
    }
}

// Función para agregar un nuevo contacto a la lista
fn agregar_contacto(contactos: &mut Vec<Contacto>) {
    let mut entrada = String::new();

    println!("Nombre:");
    io::stdin().read_line(&mut entrada).unwrap();
```

```
let nombre = entrada.trim().to_string();
entrada.clear();

// Solicita y valida el mes del cumpleaños
let mes = loop {
    let m = pedir_numero("Mes de cumpleaños (1-12):");
    if m >= 1 && m <= 12 {
        break m;
    } else {
        println!("Mes inválido. Intenta de nuevo.");
    }
};

// Solicita y valida el día del cumpleaños
let dia = loop {
    let d = pedir_numero("Día del cumpleaños:");
    if validar_dia(d, mes) {
        break d;
    } else {
        println!("Día inválido para el mes seleccionado. Intenta de nuevo.");
    }
};

// Formatea la fecha como "dd/mm"
let cumple = format!("{:02}/{:02}", dia, mes);
```

```
    println!("Teléfono:");
    io::stdin().read_line(&mut entrada).unwrap();
    let telefono = entrada.trim().to_string();
    entrada.clear();

    println!("Correo:");
    io::stdin().read_line(&mut entrada).unwrap();
    let correo = entrada.trim().to_string();

    // Se crea un nuevo contacto y se agrega a la lista en memoria
    contactos.push(Contacto {
        nombre,
        cumple,
        telefono,
        correo,
    });

    // Se guarda la lista completa en el archivo
    guardar_contactos(contactos);
    println!("Contacto agregado y guardado.");
}

// Busca un contacto por nombre (sin importar mayúsculas o minúsculas)
fn buscar_contacto(contactos: &[Contacto], nombre: &str) {
    for c in contactos {
        if c.nombre.eq_ignore_ascii_case(nombre) {
```

```
            println!("{:?}", c);
            return;
        }
    }
    println!("No se encontró el contacto.");
}

// Lista todos los contactos que cumplen años en un mes específico
fn listar_por_mes(contactos: &[Contacto], mes: &str) {
    let mes_num: u32 = match mes.trim().parse() {
        Ok(m) if m >= 1 && m <= 12 => m,
        _ => {
            println!("Mes inválido.");
            return;
        }
    };

    let mes_formateado = format!("{:02}", mes_num);
    let mut encontrados = false;

    // Filtramos y mostramos los contactos cuyo cumpleaños coincidan con el mes ingresado
    for c in contactos {
        if let Some(m) = c.cumple.split('/').nth(1) {
            if m == mes_formateado {
                println!("{:?}", c);
                encontrados = true;
```

```
            }
        }
    }

    if !encontrados {
        println!("No hay contactos con cumpleaños en ese mes.");
    }
}

// Lista todos los contactos ordenados alfabéticamente por su nombre
fn listar_ordenados(contactos: &[Contacto]) {
    let mut mapa = BTreeMap::new(); // Mapa ordenado por nombre
    for c in contactos {
        mapa.insert(c.nombre.clone(), c.clone());
    }
    for (_, contacto) in mapa {
        println!("{:?}", contacto);
    }
}

// Elimina un contacto de la lista por nombre (las mayúsculas no importan)
fn eliminar_contacto(contactos: &mut Vec<Contacto>) {
    let mut entrada = String::new();
    println!("Nombre del contacto a eliminar:");
    io::stdin().read_line(&mut entrada).unwrap();
    let nombre = entrada.trim().to_lowercase();
```

```
    let original_len = contactos.len();
    // Se retienen solamente los contactos cuyo nombre no coincida
    contactos.retain(|c| c.nombre.to_lowercase() != nombre);

    if contactos.len() < original_len {
        guardar_contactos(contactos);
        println!("Contacto eliminado correctamente.");
    } else {
        println!("No se encontró ningún contacto con ese nombre.");
    }
}



fn main() {
    let mut lista_contactos = cargar_contactos(); // Lista en memoria cargada desde el archivo

    loop {
        println!("\nMenú:");
        println!("1) Agregar contacto");
        println!("2) Consultar contacto por nombre");
        println!("3) Listar cumpleaños por mes");
        println!("4) Listar todos los contactos ordenados");
        println!("5) Eliminar contacto");
        println!("6) Salir");
```

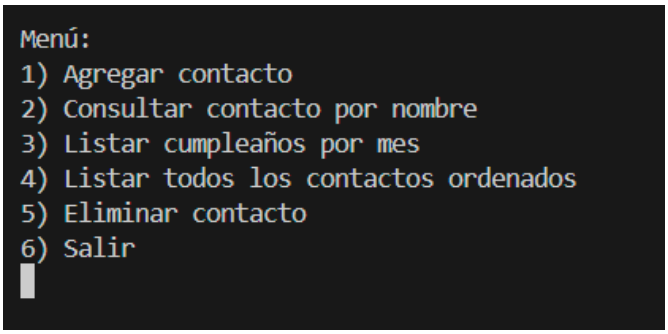
```
let mut opcion = String::new();
io::stdin().read_line(&mut opcion).unwrap();

match opcion.trim() {
    "1" => agregar_contacto(&mut lista_contactos),
    "2" => {
        let mut nombre = String::new();
        println!("Nombre del contacto:");
        io::stdin().read_line(&mut nombre).unwrap();
        buscar_contacto(&lista_contactos, nombre.trim());
    }
    "3" => {
        let mut mes = String::new();
        println!("Número del mes (1-12):");
        io::stdin().read_line(&mut mes).unwrap();
        listar_por_mes(&lista_contactos, mes.trim());
    }
    "4" => listar_ordenados(&lista_contactos),
    "5" => eliminar_contacto(&mut lista_contactos),
    "6" => {
        guardar_contactos(&lista_contactos);
        println!("Contactos guardados correctamente. Saliendo...");
        break;
    }
    _ => println!("Opción no válida"),
}
```

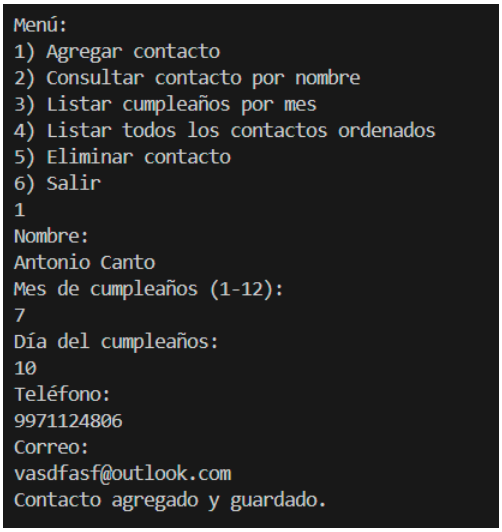
```
    }
}
```

Capturas de pantalla del código en ejecución

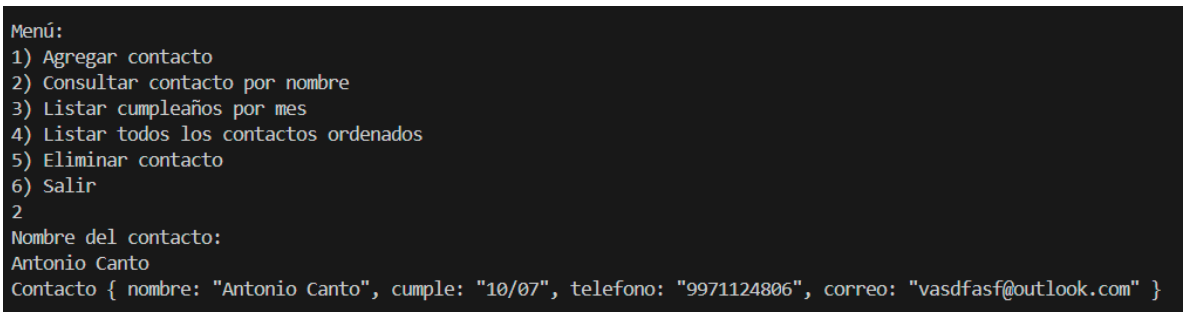
Menú de opciones:



Agregar contacto:



Consultar contacto por nombre:



Listar los contactos por mes de cumpleaños:

```
Menú:
1) Agregar contacto
2) Consultar contacto por nombre
3) Listar cumpleaños por mes
4) Listar todos los contactos ordenados
5) Eliminar contacto
6) Salir
3
Número del mes (1-12):
7
Contacto { nombre: "Antonio Canto", cumple: "10/07", telefono: "9971124806", correo: "vasdfasf@outlook.com" }
```

Listar todos los contactos ordenados alfabéticamente por nombre:

```
Menú:
1) Agregar contacto
2) Consultar contacto por nombre
3) Listar cumpleaños por mes
4) Listar todos los contactos ordenados
5) Eliminar contacto
6) Salir
4
Contacto { nombre: "Angel", cumple: "10/05", telefono: "943524", correo: "angel13411" }
Contacto { nombre: "Antonio Canto", cumple: "10/07", telefono: "9971124806", correo: "vasdfasf@outlook.com" }
Contacto { nombre: "Mauricio", cumple: "14/04", telefono: "992349234243", correo: "mauricoi3243134" }
Contacto { nombre: "Rafa", cumple: "12/12", telefono: "3441312", correo: "rafasdfadf" }
```

Eliminar contacto y verificación de que se haya eliminado correctamente:

```
Menú:
1) Agregar contacto
2) Consultar contacto por nombre
3) Listar cumpleaños por mes
4) Listar todos los contactos ordenados
5) Eliminar contacto
6) Salir
5
Nombre del contacto a eliminar:
Antonio Canto
Contacto eliminado correctamente.

Menú:
1) Agregar contacto
2) Consultar contacto por nombre
3) Listar cumpleaños por mes
4) Listar todos los contactos ordenados
5) Eliminar contacto
6) Salir
4
Contacto { nombre: "Angel", cumple: "10/05", telefono: "943524", correo: "angel13411" }
Contacto { nombre: "Mauricio", cumple: "14/04", telefono: "992349234243", correo: "mauricoi3243134" }
Contacto { nombre: "Rafa", cumple: "12/12", telefono: "3441312", correo: "rafasdfadf" }
```