**Московский государственный технический
университет им. Н.Э. Баумана**

Факультет «Информатика и системы управления»
Кафедра ИУ5 «Системы обработки информации и управления»

Курс «Парадигмы и конструкции языков программирования»
Отчет по Домашнему заданию
«Система владения и заимствования памяти в языке программирования Rust»

Выполнил:
студент группы ИУ5-36Б
Ковалев Е.А.

Проверил:
преподаватель каф. ИУ5
Гапанюк Ю.Е.

Москва, 2024 г.

Задание:

1. Выберите язык программирования (который Вы ранее не изучали) и (1) напишите по нему реферат с примерами кода или (2) реализуйте на нем небольшой проект (с детальным текстовым описанием).
2. Реферат (проект) может быть посвящен отдельному аспекту (аспектам) языка или содержать решение какой-либо задачи на этом языке.
3. Необходимо установить на свой компьютер компилятор (интерпретатор, транспилятор) этого языка и произвольную среду разработки.
4. В случае написания реферата необходимо разработать и откомпилировать примеры кода (или модифицировать стандартные примеры).
5. В случае создания проекта необходимо детально комментировать код.
6. При написании реферата (создании проекта) необходимо изучить и корректно использовать особенности парадигмы языка и основных конструкций данного языка.
7. Приветствуется написание черновика статьи по результатам выполнения ДЗ. Черновик статьи может быть подготовлен группой студентов, которые исследовали один и тот же аспект в нескольких языках или решили одинаковую задачу на нескольких языках.

Rust — это современный язык программирования, активно используемый для создания безопасных, высокопроизводительных приложений. Одной из самых заметных особенностей языка является его система управления памятью, основанная на концепциях владения (ownership), заимствования (borrowing) и заимствования с изменением (mutable borrowing). Эта система позволяет избежать множества распространённых ошибок работы с памятью, таких как утечки, висячие указатели и гонки данных, которые часто возникают в языках C и C++.

Цель данного реферата — рассмотреть механизм работы системы владения и заимствования в языке Rust, а также проведение сравнительного анализ с подходами в C и C++, чтобы понять, как Rust решает проблемы, связанные с управлением памятью.

Основы системы владения и заимствования в Rust

Язык Rust использует несколько принципов для эффективного управления памятью: владение, заимствование и ссылки. Эти концепции обеспечивают безопасность памяти без необходимости сборщика мусора, который часто является источником накладных расходов в других языках программирования.

Владение (Ownership)

Каждый ресурс в Rust, будь то переменная или объект, имеет владельца. Когда объект выходит из области видимости, его память автоматически освобождается. Владение является основой системы управления памятью в Rust, предотвращая утечки памяти и другие ошибки.

Пример:

```rust
fn main() {
    let s1 = String::from("Текстовая строка"); // s1 владеет строкой
    println!("{}", s1); // Используем строку через s1
}
```

```
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/DZ`
Текстовая строка
```

```rust
fn main() {
    {
        let s1 = String::from("Текстовая строка");
    }
    println!("{}", s1); // По завершению блока s1 выходит из области видимости
}
```

```
   Compiling DZ v0.1.0 (/Users/111ey/Desktop/rust/DZ)
error[E0425]: cannot find value `s1` in this scope
 --> src/main.rs:5:20
  |
5 |     println!("{}", s1); // По завершению блока s1 выходит из области видимости
  |                    ^^
  |
help: the binding `s1` is available in a different scope in the same function
 --> src/main.rs:3:13
  |
3 |         let s1 = String::from("Текстовая строка");
  |             ^^

For more information about this error, try `rustc --explain E0425`.
error: could not compile `DZ` (bin "DZ") due to 1 previous error
```

В этом примере переменная `s1` владеет объектом `String`. Когда `s1` выходит из области видимости, память освобождается автоматически. Это позволяет избежать утечек памяти, которые могут возникнуть в языках, требующих ручного управления памятью.

Заимствование (Borrowing)

Rust позволяет переменным заимствовать данные с помощью ссылок. Это позволяет избежать копирования данных и эффективно работать с ними, не передавая владение. Неизменяемое заимствование (immutable borrowing) позволяет нескольким частям программы читать данные, но не изменять их. Изменяемое заимствование (mutable borrowing) позволяет одной части программы изменять данные, но гарантирует, что никто другой не может изменить эти данные в это время.

Пример неизменяемого заимствования:

```rust
fn main() {
    let s1 = String::from("Текстовая строка");
    let s2 = &s1; // s2 — неизменяемая ссылка на s1
    println!("s1: {}", s1);
    println!("s2: {}", s2);
    // Все ссылки на s1 безопасны для чтения
}
```

```
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/DZ`
s1: Текстовая строка
s2: Текстовая строка
```

```rust
fn main() {
    let s1 = String::from("Текстовая");
    let s2 = &s1;
    s2.push_str(" строка"); //Выдаст ошибку при изменении
    println!("s2: {}", s2);
}
```

```
   Compiling DZ v0.1.0 (/Users/111ey/Desktop/rust/DZ)
error[E0596]: cannot borrow `*s2` as mutable, as it is behind a `&` reference
 --> src/main.rs:4:5
  |
4 |     s2.push_str(" строка"); //Выдаст ошибку при изменении
  |     ^^ `s2` is a `&` reference, so the data it refers to cannot be borrowed as mutable
  |
help: consider changing this to be a mutable reference
  |
3 |     let s2 = &mut s1;
  |              +++

For more information about this error, try `rustc --explain E0596`.
error: could not compile `DZ` (bin "DZ") due to 1 previous error
```

В этом примере `s2` является ссылкой на `s1`. Мы можем безопасно читать значение (но не изменять) из обеих переменных. При этом `s1` остаётся владельцем строки, и память будет освобождена, когда `s1` выйдет из области видимости.

Пример изменяемого заимствования:

```rust
fn main() {
    let mut s1 = String::from("Текстовая");
```

```
    let s2 = &mut s1;  // s2 — изменяемая ссылка на s1
    s2.push_str(" строка");
    println!("{}", s2); // Строка теперь изменена
    // После завершения работы с s2, s1 снова доступен для использования
}
```

```
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/DZ`
Текстовая строка
```

```
fn main() {
    let mut s1 = String::from("Текстовая");
    let s2 = &mut s1;  // s2 — изменяемая ссылка на s1
    let s3 = &mut s1; // Выдаст ошибку при создании второй ссылки
    s2.push_str(" строка");
}
```

```
error[E0499]: cannot borrow `s1` as mutable more than once at a time
 --> src/main.rs:4:14
  |
3 |     let s2 = &mut s1;  // s2 — изменяемая ссылка на s1
  |              -------- first mutable borrow occurs here
4 |     let s3 = &mut s1; // Выдаст ошибку при создании второй ссылки
  |              ^^^^^^^^ second mutable borrow occurs here
5 |     s2.push_str(" строка");
  |     -- first borrow later used here
```

В этом примере переменная `s1` изменяется через ссылку `s2`. Компилятор гарантирует, что на `s1` может быть только одна изменяемая ссылка, что предотвращает состояние гонки, если бы несколько частей программы пытались одновременно изменить данные.

Ссылки (References)

Ссылки — это способ заимствования данных. Rust имеет два типа ссылок:

- Неизменяемые ссылки (`&T`) — позволяют только читать данные.
- Изменяемые ссылки (`&mut T`) — позволяют изменять данные, но гарантируют, что данных не изменяют другие части программы одновременно.

Пример с двумя неизменяемыми ссылками:

```
fn main() {
    let s1 = String::from("Строка");
    let s2 = &s1;  // неизменяемая ссылка на s1
    let s3 = &s1;  // ещё одна неизменяемая ссылка на s1
    println!("s2: {}, s3: {}", s2, s3); // Чтение данных через обе ссылки
}
```

```
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/DZ`
s2: Строка, s3: Строка
```

В этом примере обе ссылки, `s2` и `s3`, могут безопасно читать строку, но никто не может изменять её, пока ссылки существуют. Компилятор гарантирует, что только неизменяемые ссылки могут сосуществовать.

Пример с изменяемой и неизменяемой ссылкой (ошибка компиляции):

```
fn main() {
    let mut s1 = String::from("Hello");
    let s2 = &s1;  // неизменяемая ссылка
    let s3 = &mut s1;  // ошибка компиляции: не может быть одновременно неизменяеменная и изменяемая ссылка
    println!("{}", s2);
}
```

```
error[E0502]: cannot borrow `s1` as mutable because it is also borrowed as immutable
 --> src/main.rs:4:14
  |
3 |     let s2 = &s1;  // неизменяемая ссылка
  |              --- immutable borrow occurs here
4 |     let s3 = &mut s1;  // ошибка компиляции: не может быть одновременно неизменяеменная и изменяемая ссы...
  |              ^^^^^^^ mutable borrow occurs here
5 |     println!("{}", s2);
  |                    -- immutable borrow later used here
```

Этот код не скомпилировался, потому что Rust запрещает одновременно иметь изменяемую и неизменяемую ссылку на одни и те же данные, чтобы избежать гонок данных.

Преимущества системы владения и заимствования

Безопасность памяти

Rust гарантирует безопасность памяти на уровне компиляции, предотвращая ошибки, такие как:

- Использование после освобождения
- Разыменование висячих указателей;
- Двойное освобождение памяти.

Пример использования после освобождения (ошибка компиляции):

```
fn main() {
    let s1 = String::from("Hello");
    let s2 = &s1;
    drop(s1); // Память освобождена вручную
    println!("{}", s2); // Ошибка компиляции: нельзя использовать ссылку после освобождения памяти
}
```

```
   Compiling DZ v0.1.0 (/Users/111ey/Desktop/rust/DZ)
error[E0505]: cannot move out of `s1` because it is borrowed
 --> src/main.rs:4:10
  |
2 |     let s1 = String::from("Hello");
  |         -- binding `s1` declared here
3 |     let s2 = &s1;
  |              --- borrow of `s1` occurs here
4 |     drop(s1); // Память освобождена вручную
  |          ^^ move out of `s1` occurs here
5 |     println!("{}", s2); // Ошибка компиляции: нельзя использовать ссылку после освобождения пам...
  |                    -- borrow later used here
```

Компилятор Rust не позволяет использовать переменную `s2` после того, как память, на которую она ссылается, была освобождена.

Предотвращение гонок данных

Rust предотвращает гонки данных, обеспечивая, что либо существует несколько неизменяемых ссылок на данные, либо одна изменяемая ссылка. Это исключает возможность одновременной записи в память из нескольких потоков.

Пример гонки данных в многозадачной среде (ошибка компиляции):

```
use std::thread;

fn main() {
    let mut s1 = String::from("Текстовая");
    let s2 = &mut s1; // изменяемая ссылка

    let handle = thread::spawn(move || {
        s2.push_str(" строка"); // изменение данных в другом потоке
    });

    handle.join().unwrap();
    println!("{}", s1); // Ошибка компиляции: доступ к данным из разных потоков
}
```

```
error[E0597]: `s1` does not live long enough
  --> src/main.rs:5:14
   |
4  |         let mut s1 = String::from("Текстовая");
   |             ------ binding `s1` declared here
5  |         let s2 = &mut s1; // изменяемая ссылка
   |                  ^^^^^^^ borrowed value does not live long enough
6  |
7  |         let handle = thread::spawn(move || {
   |  ____________________-
8  | |           s2.push_str(" строка"); // изменение данных в другом потоке
9  | |       });
   | |______- argument requires that `s1` is borrowed for `'static`
...
13 |     }
   |     - `s1` dropped here while still borrowed
```

Этот код не скомпилировался, поскольку переменная `s1` изменяется в одном потоке через ссылку `s2`, и Rust не позволяет передавать изменяемую ссылку в другой поток, чтобы избежать гонки данных.

Отсутствие накладных расходов на сборщик мусора

В отличие от языков, таких как Java или Python, Rust не использует сборщик мусора. Вместо этого управление памятью происходит на этапе компиляции. Это исключает дополнительные накладные расходы на работу с мусором и повышает производительность.

Сравнение с C и C++

В C и C++ программисты вручную управляют памятью, используя указатели и функции для выделения и освобождения памяти. Ошибки в этих операциях могут привести к утечкам памяти, повреждению данных и другим проблемам.

Пример на C

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    char* s1 = (char*) malloc(14 * sizeof(char)); // выделение памяти
    strcpy(s1, "Hello, world!"); // копирование строки
    printf("%s\n", s1);

    free(s1); // явное освобождение памяти
}
```

В этом примере программист обязан вручную управлять памятью, вызывая `malloc` и `free`. Это может привести к ошибкам, таким как забывание вызова `free` или двойное освобождение памяти.

Пример на C++ с умными указателями

```cpp
#include <iostream>
#include <memory>

int main() {
    auto s1 = std::make_unique<std::string>("Hello, world!"); // умный указатель
    std::cout << *s1 << std::endl; // доступ через умный указатель
    // Память освобождается автоматически
}
```

C++ предлагает умные указатели, которые автоматизируют управление памятью, но они не защищают от всех возможных ошибок, таких как гонки данных или неконтролируемые изменения состояния в многозадачных приложениях.

Преимущества и недостатки системы владения Rust

Преимущества

- Безопасность на уровне компиляции: Компилятор Rust предотвращает ошибки работы с памятью, такие как утечки и гонки данных.
- Высокая производительность: Поскольку нет необходимости в сборщике мусора, Rust может работать быстрее, чем языки с автоматическим управлением памятью.
- Гарантии безопасности многозадачности: Rust предотвращает гонки данных, что делает его идеальным для многозадачных приложений.
- Предсказуемость: Строгая типизация и контроль над ресурсами обеспечивают предсказуемое поведение программы.

Недостатки

- Кривая обучения: Для новичков концепции владения и заимствования могут быть сложными для понимания.
- Ограниченная гибкость: В некоторых случаях разработчики могут захотеть больше контроля над управлением памятью, чего невозможно достичь с помощью Rust.

Система владения и заимствования в Rust является важным инструментом для обеспечения безопасности памяти и предотвращения ошибок. В отличие от C и C++, Rust не требует явного управления памятью, что снижает вероятность ошибок, таких как утечки или гонки данных. Несмотря на сложность освоения концепций владения, Rust предлагает значительные преимущества в плане безопасности и производительности, что делает его отличным выбором для системного программирования и разработки многозадачных приложений.