Rust Learning Repository

Getting started

Name

The project name is "Rust Learning Repository".

Description

The "Rust Learning Repository" is a comprehensive collection of Rust programming examples and tutorials designed to facilitate the learning process for individuals interested in mastering the Rust programming language.

Key areas covered in the repository include:

1. Theoretical Concepts:
    - Explanation of fundamental concepts such as Heap, Stack, and the absence of a Garbage Collector in Rust.
    - Comparison with analogous concepts in C to highlight differences and similarities.
2. Performance and Memory Management:
    - Exploration of performance using profiling methods in Rust.
3. Safety Features:
    - Detailed examination of Rust's safety features, including:
        - Ownership: Understanding Rust's ownership model and its impact on memory management.
        - Types: Exploring advanced type system features for ensuring code safety.
        - Threads: Demonstrating the use of Arc and Mutex for thread safety in concurrent programming scenarios.

Installation

Rust Installation

To use this project, you need to have Rust installed on your system. If you're using Linux or macOS, you can easily install Rust via `rustup`. Follow these steps to install `rustup`:

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

This command downloads a script and starts the installation of the `rustup` tool, which installs the latest stable version of Rust. You might be prompted for your password during the installation process. If the installation is successful, you will see the following message:

```
Rust is installed now. Great!
```

Additionally, you will need a linker, which is a program that Rust uses to join its compiled outputs into one file. Most systems already have a linker installed. However, if you encounter linker errors, you may need to install a C compiler, as it typically includes a linker.

On macOS, you can get a C compiler by running:

```
xcode-select --install
```

Package Installation

Make sure to install any required packages, such as profilers, before running the project.

Usage

Building and Running

To build and run individual files in the project, follow these steps:

1. Clone the repository to your local machine.
2. Navigate to the project directory.
3. Run the following command to build a specific file, replacing `<file_name>` with the name of the Rust file without the `.rs` extension:

```
cargo build --bin <file_name>
```

4. After the build completes successfully, you can run the compiled binary using:

```
cargo run --bin <file_name>
```

Replace `<file_name>` with the name of the Rust file you want to build and run. Here are the available files:

- ownership
- ownership2

- types
- threads
- arc
- sendsync
- mutex
- mutexwithoutarc
- arcneedssync
- arcneedstypes

For example, to build and run the `ownership.rs` file, you would use:

```
cargo build --bin ownership
cargo run --bin ownership
```

Additionally, you can add new binaries to the `cargo.toml` file with a specified name for easier referencing. Simply add the following lines to `cargo.toml`, replacing `<name>` with the desired name and `<path>` with the path to the Rust file:

```
[[bin]]
name = "<name>"
path = "<path>"
```

For example:

```
[[bin]]
name = "custom_binary"
path = "src/custom_file.rs"
```

Then, you can build and run this custom binary using:

```
cargo build --bin custom_binary
cargo run --bin custom_binary
```

Adding Dependencies

If you need to add dependencies to your project, you can do so by editing the `Cargo.toml` file. Simply add the dependency under the `[dependencies]` section, specifying the name and version number. For example:

```
[dependencies]
crossbeam = "0.8"
```

C Compilation

GCC Installation

If you're working with C code, you'll need to have the GCC compiler installed on your system. You can install GCC using your system's package manager. For example, on Ubuntu, you can install GCC with the following command:

```
sudo apt-get update
sudo apt-get install build-essential
```

Compiling and Running

Once GCC is installed, you can compile your C code using the `gcc` command. Here's a basic example:

```
gcc -o program program.c
```

Replace `program.c` with the name of your C source file. After compilation, you can run the compiled executable:

```
./program
```

Additional Tools

Htop Installation

If you want to monitor system resources while running your programs, you can install `htop` on Linux. Here's how to install it:

```
sudo apt-get update
sudo apt-get install htop
```

After installation, you can run `htop` from the command line to view real-time system statistics.

Profiling

What is profiling?

The profiling is a dynamic program analysis, that measures the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls.

The goal of profiling is to detect memory leaks and other inefficiencies in the code. It’s an important step in the optimization phase of software development where developers aim to enhance the speed, the memory usage and the overall performance of the software.

Memory profiling with massif in valgrind

Massif, a tool within Valgrind, facilitates memory profiling by scrutinizing memory usage throughout program execution, capturing snapshots, and providing detailed analysis.

To generate a Massif file, begin by compiling your program:

```
rustc -O -g salaries.rs
```

Next, execute the following command :

```
Valgrind --tool=massif salaries
```

Finally, to format the Massif file for analysis, utilize the following command :

```
Ms_print massif.out.(massif file number)
```

Replace "(massif file number)" with the appropriate file number generated during the execution of your program.

Flame graph

A flame graph provides a visual representation of where a program allocates its memory. It illustrates the combination of all stack traces leading to active memory allocations at a specific point in time.

In a flame graph, each stack trace is depicted as a column of boxes, with each box representing a function call within that stack.

The y-axis denotes the stack depth. A special row represents the root, followed by rows representing top-level function calls that triggered memory allocations, subsequent rows representing functions called by those top-level calls, and so forth. The function furthest from the root in any vertical slice represents the one directly responsible for memory allocation, while the boxes leading back to the root outline the full call stack leading to that allocation.

The width of each box corresponds to the amount of memory allocated by the associated function call or its descendants. Wider boxes indicate more significant memory allocations relative to narrower ones.

How to generate a flame graph

To create a flame graph, you'll need to run a Dockerfile, which sets up an Ubuntu server with Rust installed.

Once inside your container, install Flamegraph by executing the following command:

```
cargo install flamegraph
```

Next, compile your file:

```
rustc -O -g salaries.rs
```

Now, generate the flame graph :

```
cargo flamegraph --bin salaries
```

This will produce an SVG file. To view it, transfer the file from your container to your desktop :

```
docker cp [container_id]:/app/src/flamegraph.svg [local_path]
```

Replace '[container_id]' with your container ID, and '[local_path]' with the desired path on your desktop.

Authors and acknowledgment

We would like to express our gratitude to the following individuals for their contributions to this project:

- **Rafael Cardoso**
- **Zotrim UKA**
- **Térence Laurent**

Additionally, for the section on Arc and Mutex in Rust, we drew inspiration from the article written by **Piotr Sarnacki** on May 28, 2022, titled "Arc and Mutex in Rust" available at itsallaboutthebit.com/arc-mutex/ (https://itsallaboutthebit.com/arc-mutex/). Their insights and explanations have greatly enriched our understanding of these topics.