Proposal: Comparing Rust to C/C++ using BFS

Mahbod Asnaashari	Li Ze Choo	Veronika Ebenal	Amir Sangi	Zhishan Liu
d8d1b	l9h0b	r5v9a	j2q0b	j7n8
45189157	33924151	30629133	13105144	40067126
d8d1b@ugrad.cs.ubc.ca	l9h0b@ugrad.cs.ubc.ca	r5v9a@ugrad.cs.ubc.cs	j2q0b@ugrad.cs.ubc.cs	j7n8@ugrad.cs.ubc.cs

INTRODUCTION

For this project, we have chosen project type four: creating a substantial program in a potentially useful but esoteric language to illustrate the particular value of the language. More specifically, we will be creating equivalent programs in Rust and C/C++ which will both search for the optimal route in a maze or graph using BFS. This will be done to show that Rust allows for more efficient memory management compared to C/C++.

Language Justification

Rust is a relatively new language which has gained much praise for being user-friendly and safe alternative to C/C++. Rust shares several similarities with C++, in that it has its own version of object classes, known as traits. It also follows the same programming idiom for resource allocation and object lifetime known as Resource Acquisition Is Initialization (RAII). The differences lie in the fact that Rust incorporates a range of elegant memory handling techniques that prevent common issues, such as null pointers, dangling pointers, or data races in safe code. It is a language designed to be memory safe. Despite this overhead, Rust is still able to compete with C++ in speed, making it an excellent language to explore.

Comparison Metrics

We will attempt to show the benefits of Rust's memory handling, running time optimizations, and user-friendly error handling by comparing several graph search programs in C/C++ and with Rust. Our expectation is to show that Rust will perform better in memory safety while maintaining program efficiency over C/C++. When given buggy, badly-formed code, C compiles and might return an incorrect result, while Rust returns errors that help pinpoint memory leaks and dangling pointers. Memory management of programs in C/C++ will be profiled using Valgrind while program written in Rust will be profiled using the *heapsize* package: https://crates.io/crates/heapsize.

Program Justification

A BFS traversal of a graph uses b^m memory. This means that even with a moderately sized graph, memory storage can exponentially grow large enough that the differences in memory handling between Rust and C can be quantified. This makes BFS traversal preferable to other types of graph traversal, such as DFS, which are less memory dependant.

OUTLINE FOR PROJECT MILESTONES

We will be aiming to reach the 100% milestone for this project. While this proposal as a whole outlines our intended goal for this project, the project plan below details a low-risk approach we can use to get to the 80% milestone.

Comprehensive Project Plan

- To begin with the project we will first focus on adopting the agile development process to make sure that we can effectively function as a team. Through this we could plan to have daily scrum meetings that could happen online over slack so that we can quickly see progress and/or impediments that we may need to resolve.
- Then, we will develop a plan for each specific detail of creating the program in Rust and C using a format of that similar to user stories, however, not as formal or concise.
- After a detailed breakdown of the constructions of the program is built, we will begin by giving each team member different specific tasks to work on with weekly goals.
- With the implementation of these processes, we will be able to efficiently and steadily work towards the end goal of our project.
- Finally, we plan on at least going to one tutorial or office hour as a team to check our progress/get help to keep our progress on track.

Further Research

Mozilla Tech: Rust and the Future of Systems Programming

https://medium.com/mozilla-tech/rust-and-the-future-of-systems-programming-b75fba746910

The videos on this site give a quick introduction on the benefits of using Rust. One of the great advantage is that Rust ensures the safety of the code. It does error checking when compiling, and immediately alters programmers that the code is incorrect, unlike some other languages that crash when the program is executed. To demonstrate this we can write a program which contains bugs that are hard to spot, and let the compiler check the mistakes and return specific information about the incorrect code.

The site also mentions how Rust is powerful when writing multi-threading programs: the error checking on memory usage also checks the ownership of that block of memory. This allows programmers to know which thread is using a given memory block to avoid multiple threads accessing the same chunk and consequently breaking the program. This makes it easier to write code that runs parallel across multiple cores.

We may as well write our BFS program using the benefit of multi-threading in order to aid in understanding why Rust is thread safe and to write a good program using parallelism. The following blogs provide detailed interpretation on how Rust ensures the safety of using threads:

1. *Defaulting to Thread-Safety: Closures and Concurrency* https://huonw.github.io/blog/2015/05/defaulting-to-thread-safety/
2. *How Rust Achieves Thread Safety* https://manishearth.github.io/blog/2015/05/30/how-rust-achieves-thread-safety/
3. *Finding Closure in Rust* https://huonw.github.io/blog/2015/05/finding-closure-in-rust/
4. Fearless Concurrency with Rust https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html
5. Why Rust's Ownership/Borrowing is Hard http://softwaremaniacs.org/blog/2016/02/12/ownership-borrowing-hard/en/

Rust Learning

https://github.com/ctjhoa/rust-learning

This contains a collection of resources to learn about Rust. To focus on learning the semantic of Rust, we can find helpful blogs related to the language under this category: https://github.com/ctjhoa/rust-learning#language-stuff

Poster

Our poster will incorporate four main ideas in order to aid our ability to efficiently and thoroughly demonstrate the value of our project: that is, the merits of using Rust over C/C++.

1. *BFS Traversal*

We will first quickly go over the program which we designed. Our poster will have a visual representation of BFS traversal.

2. *Memory Handling*

The poster will graphically represent the memory handling in terms of the heapsize used, blocks freed, and blocks allocated. The graphs for C/C++ and Rust will be superimposed in order to efficiently represent the differences between them.

3. *Runtime Optimization*

The poster will show how Rust maintains program efficiency while maintaining memory safety by graphically comparing the runtimes of the BFS traversal in C/C++ and in Rust.

4. *Error Handling*

The poster will show how Rust does not permit the presence of null pointers, dangling pointers, or data races. This will be shown through programs containing the aforementioned bugs and how these programs compile in C/C++ but throws errors in Rust when compiled.

STARTING POINT DOCUMENTS

To begin our background research report we will need to obtain a variety of information relating to both logistics and theory. The documents below serve as sources which we can use not only to initially familiarize ourselves with the concepts addressed in our project, but throughout the project as a whole.

Language Documentation: The Rust Standard Library

https://doc.rust-lang.org/std/

This documentation will be a useful reference throughout the project, particularly when first familiarizing ourselves with Rust in the project proof-of-concept, as well as during the actual code implementation of the project. It contains definitions of primitive types, modules, and macros in the Rust std

crate. It also has links to documentation of other Rust crates if we use them.

Blog: An Overview of Memory Management in Rust
http://pcwalton.github.io/blog/2013/03/18/an-overview-of-memory-management-in-rust/

This blog uses easy-to-understand examples in order to demonstrate the manual memory management used in Rust. It explains smart pointers and references, and explicitly describes where they logistically differ from C.

Blog: Rust Memory Management for C/C++ Programmers
http://blog.zgtm.de/1

This blog again discusses smart pointers and memory management in Rust. However, it seems considerable more substantial and goes into detail on heaps, variables, functions, and how everything relates to memory logistically.

Blog: Rust for C++ Programmers
https://aminb.gitbooks.io/rust-for-c/content/

Many of the basic-to-intermediate aspects of programming are discussed step by step. Examples in C++ are directly translated to Rust so that a programmer versed in C++ can understand where exactly Rust differs from it and where using similar syntax can be appropriate.

Ebook: The Rust Programming Language
https://doc.rust-lang.org/book/second-edition/

A cohesive overview on Rust as a whole. This book covers both practical lessons on topics such as structs and error handling, as well as theory-based topics such as “Is Rust an Object-Oriented Programming Language?”

The lessons on Rust use “small, focused examples that build on each other to demonstrate how to use various features of Rust as well as how they work behind the scenes” (Introduction). Therefore, this resource will once again be valuable on getting a solid grasp on writing and understanding programs in Rust as we work on the project.

Crates
https://crates.io/

We will likely use crates at some point in this project. For example: we will likely use the heapsize package to profile our program for memory usage etc. This documentation will allow us to find and download appropriate packages which we may need.