Computer Applications
CA341 Assignment 1 2023/24
Language Comparison

Comparing COBOL and Rust

08/11/2023

Pair Members:

Hephzibah Bode-Favours
21344221
hephzibah.bodefavours2@mail.dcu.ie

Nino Candrilic
21764295
nino.candrlic2@mail.dcu.ie

Introduction

Languages:

For our assignment the languages we chose were COBOL and rust. COBOL is an old unique language not known by many and is really only used by banks so we thought it would be a good way to learn a rare coding language. Rust is another lesser known language not as much as COBOL though, it has high software safety and performance so we thought it would be a useful language to learn.

Urls:

https://gitlab.computing.dcu.ie/bodefah2/euler/-/tree/CA341-project?ref_type=heads

Language Summary:

Rust is a modern systems programming language that aims to provide memory safety, concurrency, and performance. It was developed by Mozilla and first released in 2010. Rust's main goal is to eliminate common programming errors, such as null pointer dereferences and data races, by enforcing strict compile-time checks.

COBOL (Common Business-Oriented Language) is a high-level programming language that was developed in the late 1950s. It was designed specifically for business applications and is known for its readability and English-like syntax. COBOL was widely used in the past for mainframe and large-scale business systems, and it still plays a significant role in legacy systems today.

Pros and Cons:

Rust

Pros	Cons
Memory Safety: Rust's ownership system and borrowing rules help prevent common programming errors such as null pointer dereferences, dangling pointers, and data races, making it a memory-safe language.	**Learning Curve:** Compared to certain other programming languages, Rust has a higher learning curve, particularly for developers who are not familiar with ideas like ownership, borrowing, and lifetimes.
Performance: Without compromising on functionality, Rust offers granular control over system resources. It frequently performs similarly to languages like C and C++ and has predictable performance characteristics.	**Build Times:** When it comes to large projects, Rust's compilation times may be longer than those of some other languages. To improve compilation times, though, continued efforts are being made.
No Garbage Collection: Rust manages resources in a more predictable and deterministic manner by achieving memory safety without the need for garbage collection.	**Strict Ownership Rules:** For developers used to garbage-collected languages, ownership and borrowing can be burdensome even though they offer memory protection. Ownership semantics must be carefully taken into mind.

COBOL

Pros	Cons
Readability: With a syntax that is similar to natural language, COBOL code is frequently regarded as being simple to read and comprehend. This may work to the benefit of modernizing and sustaining legacy systems.	**Dated Syntax**: Some programmers, particularly those used to more contemporary languages, feel that COBOL's syntax is archaic. Some of the characteristics of modern languages are absent from it.
Support for Legacy Systems: COBOL is still widely used in many important business applications that are written on legacy systems. Because of its durability and stability, it can be used to update and maintain current systems.	**Restricted Abstraction:** COBOL lacks higher-level abstractions seen in modern languages and only partially supports contemporary programming principles. Because of this, it might not be as appropriate for some uses.
Data Handling: COBOL is an excellent data handler, which makes it a good choice for applications requiring a lot of data processing, such as financial transactions.	**Limited Libraries and Frameworks:** The COBOL language has a smaller library and framework selection than other more contemporary languages. Developers might have to create a lot of components from the ground up.
Processing Speeds: COBOL is in essence, assembly with extraordinary quality-of-life features(more bare-metal you can hardly get). It was designed to run on systems of its time; a time where RAM capacity was still counted in Megabytes, and clock speeds barely reached a single Gigahertz.	

Comparative Analysis

Euler Questions:

Problem 1
Problem 3
Problem 4
Problem 10
Problem 100
Problem 78
Problem 114
Problem 117
Problem 160
Problem 162

These questions were chosen because after looking through euler these seemed like the most interesting questions and they also gave us opportunities to learn more about our chosen languages

Problem 1: Sum of Multiples

Rust (OO Framework)

The Rust solution defines a struct (SumOfMultiples) to hold the problem-related data and behavior in accordance with the Object-Oriented paradigm. The logic for computing the sum of multiples is contained in the struct's calculate_sum method, which acts on a struct instance. The method is called on the created instance of the struct, which is created with the given limit.
https://gitlab.computing.dcu.ie/bodefah2/euler/-/blob/CA341-project/problems/euler-1-multiples-of-3-or-5.rs/euler-1-multiples-of-3-or-5.rs?ref_type=heads

COBOL (Procedural Paradigm)

Because variables in COBOL must be defined on compile-time, all the variables are defined in the WORKING-STORAGE DIVISION. Here one may see a benefit to the way numbers are handled. The variable “N”, an 8 digit decimal number (PIC 9(8) BINARY) makes sense since the test cases are not meant to exceed 10^8. The memory taken is as much as this variable will ever need(akin to u8, u32, and u64, in Rust; promoting conscious memory allocation). Given COBOL’s goal of readability, it makes sense why data is defined this way. For someone who is not familiar with programming concepts such as primitives(and more importantly, the range of values it can represent), it makes sense to define data by what “character” is expected to be stored; if one is expecting to deal with millions, he uses a number with 7 digits.

Another interesting part of this code is the use of sections. The 01-PREAMBLE section serves as a main function place-in. It takes the input, gets the solution, and then displays it. That PERFORM in 01-PREAMBLE is actually how COBOL implements loops(for-loop in this case). Sure, that code could have been written inline, but it's easier to read if the code is abstracted away into the 02-CHECK-RANGE section. This entire section is performed on every iteration of the above PERFORM statement. As per what it does, it merely adds to the “total” all “num” which are either divisible by 3 or 5. Here we can observe further abstracting away by separating code into 02-CHECK-IF-DIV-N paragraphs. I read once that COBOL was designed to be self-documenting, and truly, between sectioning/paragraphing off code, and the verbosity of its statements, there was rarely ever any need for comments. The commenting and code writing are near to being one and the same in COBOL, as opposed to Rust where they have no functional relevance.
https://gitlab.computing.dcu.ie/bodefah2/euler/-/tree/CA341-project/problems/euler-1-multiples-of-3-or-5.cbl?ref_type=heads

Problem 4: Largest Palindrome Product

Rust (OO Framework)

The Rust approach defines two structs (PalindromeChecker and PalindromeFinder) in accordance with the Object-Oriented paradigm. It contains the reasoning behind examining palindromes and determining which palindrome product has the greatest amount of digits. Code is clearly organized and roles are defined through the usage of structs and methods.
https://gitlab.computing.dcu.ie/bodefah2/euler/-/blob/CA341-project/problems/euler-4-largest-palindrome-product.rs/euler-4-largest-palindrome-product.rs?ref_type=heads

COBOL (Procedural Paradigm)

The 00-MAIN section serves the same purpose as 01-PREAMBLE did in previous code. In this example, we may observe what a nested for-loop equivalent looks like. Performing 01-A-SELECTOR section, which in itself only performs the 02-B-SELECTOR, does achieve the behavior of a nested for-loop, but it muddies up the code by: a) abstracting away a part of a composite structure - a nested loop is a composite of two loops - thus forcing the reader to misinterpret the code. If you are looking at the PERFORM in 00-MAIN, you see a for-loop; if instead you are looking at the 01-A-SECTION you see that some line in the entire division is going to execute a for-loop. There is no way of knowing this is a nested-loop without reading the entire code which programmers tend to skim due to sheer volumes of text involved. It is interesting to observe that such a problem would not occur to someone programing in rust, because it's obvious that code blocks ought to be nested(and there is a reason why it is unwise to do so in COBOL which is elaborated upon in the next problem), whist a COBOL programer may arrive at this bad practice naturally. Bad practice indeed, for COBOL does support nested PERFORMs, which is why, if I were to ever tidy up this code, I would apply Rust reasoning, and simply nest the performs.

With loops iterating over values of interest, the 02-B-SELECTOR section verifies if the product is a palindrome. And this is done in a language specific way. For if you turn your attention to the 03-REVERSE-NUMBER section, you will notice that all that is being done is in essence, moving data between variables, and a bit of multiplication. It is exploiting the fact that moving a value into (a smaller or) a single digit variable isolates the digit. In the case of TABLEs, an 18 digit number can be MOVEd into a table with 18 occurrences of single digit numbers, each individually accessible. These design decisions make sense, given that COBOL was intended for processing large volumes of data.
https://gitlab.computing.dcu.ie/bodefah2/euler/-/blob/CA341-project/problems/euler-4-largest-palindrome-product.cbl/euler-4.cbl?ref_type=heads

Problem 160: Factorial Trailing Digits

Rust (OO Framework)

The Rust solution makes use of the Object-Oriented paradigm and has a struct called Calculator that has computation-related methods. To represent the calculations succinctly, iterators and functional programming principles are used. After the struct instance is constructed, the result is obtained by calling its method.

https://gitlab.computing.dcu.ie/bodefah2/euler/-/blob/CA341-project/problems/euler-162-hexadecimal-numbers.rs/euler-162-hexadecimal-numbers.rs?ref_type=heads

COBOL (Procedural Paradigm)

This problem is solved by utilizing recursion, and surprisingly enough, COBOL supports it. This is a good opportunity to explain how COBOL programs and subprograms interact. For one, programs are standalone, while subprograms require some input. The expected parameters are defined in the LINKAGE section and are then utilized by adding “USING variable” at the end of the PROCEDURE division. A variable can be used “BY REFERENCE”(allowing the subprogram to modify program variables), or “BY VALUE”(doing operations without touching program memory). The return value of the subprogram is placed into the RETURN-CODE special register. To finish execution, GOBACK is called to return to the program up the stack. Regarding, the program, it CALLs the subprogram, passing it the values(numbers, strings, or addresses). The program name is defined by a PROGRAM-ID right below the IDENTIFICATION DIVISION. For recursive calls, the PROGRAM-ID must say that it IS RECURSIVE.

https://gitlab.computing.dcu.ie/bodefah2/euler/-/blob/CA341-project/problems/euler-162-hexadecimal-numbers.cbl/euler-162-hexadecimal-numbers.cbl?ref_type=heads

Problem 117: Red, Green, and Blue Tiles

Rust (OO Framework)

The Rust approach defines a struct (WaysCal) in accordance with the Object-Oriented paradigm. It summarizes the reasoning behind figuring out how many steps are needed to get a given score. The result can be computed and initialized using methods in the struct. It stores the intermediate results in an array.

https://gitlab.computing.dcu.ie/bodefah2/euler/-/blob/CA341-project/problems/euler-117-red-green-and-blue-tiles.rs/euler-117-red-green-and-blue-tiles.rs?ref_type=heads

COBOL (Procedural Paradigm)

This problem has little innovative functionality so it is instead written such as to demonstrate a unique difference between COBOL and Rust. COBOL was intended to be programmed on a physical medium(punch-cards), while Rust was intended to be written virtually. This is a unique characteristic of COBOL. Consider, a punch-card has limited volume; it is a piece of paper and only 72 characters(referred to as columns in COBOL) could fit. To this day, IBM mainframes with the IBMCOB dialect cannot compile a script if a line has more than 72 characters. Consider further, a paper slip is no different than any other. How does a programmer know which slip belongs where in the program? It has to be numbered; this number takes the first 6 characters of the line; typically starting from 000100, and incrementing by 100. The 8th character is a comment flag. If "*" is present, the entire line is ignored. In case of wanting to comment on the line "000100", the line would start with "000101*", and increment by 1, as many times as there are lines. Finally, the actual code can be written. The variable name length limit is derived from similar reasons. This makes COBOL unique in that the compiler will not allow compilation of files with names larger than the name character limit. This is because COBOL treats file names, and program names the same. Calling a program is calling the file, and so the name limit exits. Naturally, COBOL dialects have evolved from their humble beginnings, yet even in Freeform dialect(A variant of COBOL which drops a good deal of these limits) the "feel-of-paper" is rather evident. It does have a result of making code look compact, as opposed to the unintelligible mess an HTML file can turn into.

https://gitlab.computing.dcu.ie/bodefah2/euler/-/tree/CA341-project/problems/euler-117-red-green-and-blue-tiles.cbl?ref_type=heads

Observations:

- With the use of sections, paragraphs and imperative statements, COBOL relies on explicit control flow. CALL is used for subprograms(allows passing parameters), GO TO for moving the instruction pointer to a desired line(section, or paragraph), and PERFORM which moves the pointer to the beginning of the section/paragraph, and executes each statement within(used to achieve pseudo-methods, for-loops, and while-loops). COBOL will, in absence of directions, continue increasing the instruction pointer until it reaches the end of the PROCEDURE division. This allows for "bleed" between sections/paragraphs; useful when kept under control.
- Rust improves code organization by using structs and methods to implement encapsulation and abstraction.
- COBOL computes using imperative statements and follows a procedural model with parts.
- Rust uses iterators and functional programming ideas to express calculations in a more succinct manner.
- For computations, COBOL uses explicit control flow and procedural statements.
- COBOL does not have native bit-shifting capabilities. For COBOL(2002), a language which is in essence assembly with verbose mnemonics to not have support for an instruction which was present in Intel 8088 CPU(1979) instruction set(look for R/LSH), left this group (and other witnesses) catatonically bewildered.

Data Types:

Rust: Rust features a robust system with type inference that is statically typed. Members of structs and function arguments have their types defined explicitly.

COBOL: COBOL does not support dynamic memory allocation, and thus requires all the memory required to be known on compile-time. Types in COBOL are weak, and static. COBOL uses PICTURE clauses, which define how much memory is allocated(single-word, double-word), how many digits are expected(5 digit number, or 5 character digits, or even 2 number digits followed by 3 alpha numeric digits(yes, they can be combined)) and helps the compiler to type-match variables to functions, and calls. Otherwise, they are treated as bytes. Meaning that type-casting is not only an option, but a recommendation(Examples in the problems above). A 8 digit number may exist only to read digits from input and show them in the output("USAGE IS DISPLAY"; number as ASCII), or it may exist for fast arithmetic operations("USAGE IS COMP-4"; number as a single-word binary number)

Bindings and Scope:

Rust: Rust offers a strong mechanism for controlling scope via explicit lifetimes. Rules pertaining to ownership and borrowing protect memory.
COBOL: All variables are program scoped. Programs cannot see each other's variables. Within a program, however, all variables are global. Memory is allocated at once when the program starts, and is cleared up when the program is done(Lack of dynamic memory allocation makes memory usage predictable, thus is handled by the compiler).

Abstract Data Types:

Rust: Rust supports the creation of abstract data types through the use of structs and traits. Traits provide a way to define shared behavior.
COBOL: COBOL lacks native support for abstract data types. TABLEs (COBOL structures) are used for organizing memory to receive data from input, or format data for output by having the representative bytes be ordered as the developer desires.

(Steve Klabnik, Carol Nichols (The Rust Programme Language))
(M. K. Roy,D. Ghosh Dastidar (COBOL Programming))
(Maurice Herlihy, Nir Shavit (The Art of Multiprocessor Programming))

Insights on Chosen Languages (Rust and COBOL)

Aspect	Rust	COBOL
Paradigm	Multi-paradigm (Object-Oriented, Functional, Procedural)	Procedural
Memory Safety	Strong emphasis on safety through ownership and borrowing	Buffer overflows are a constant threat. There is no dynamic memory allocation, having the memory known on compile-time, theoretically, preventing memory leakage… unless you use OpenCOBOL; it resizes the tables(COBOL lists) automatically, thus leading to unpredictable memory usage if no fail safes are set by the programmer.
Syntax and Abstraction	Modern syntax, expressive, strong abstraction capabilities	Great verbosity in syntax. Less expressive, often taking a number of statements to implement a mathematical formula. Only forms of abstracting away code in COBOL are through subprograms, or sections and paragraphs.
Concurrency	Strong support for concurrency with ownership model	No inherent support. Requires legacy COBOL libraries. But has the capacity to be compiled with C libraries.
Community and Ecosystem	Growing and vibrant community, rich ecosystem with strong libraries	Mature, with an ecosystem that focuses on enterprise systems. Excellent documentation(IBM), but suffers from a dwindling number of users(COBOL Cowboys).

Tooling	Excellent tooling (Cargo, rustfmt) with package management	MInimal tooling. Only requires a compiler (And an IBM mainframe, if you want to do it “right”).
Error Handling	Result and Option types for explicit error handling	Error codes, special registers, conditions, and verbose semantics for common errors(“ADD a TO b ON SIZE ERROR …”).
Data Types and Safety	Strong and static typing, type inference, promotes safety	Explicit, statically defined data definitions declared in a special divisions(DATA DIVISION, WORKING-STORAGE SECTION, LINKAGE SECTION)
Object-Oriented Features	Native support for OOP with structs and traits	Purely procedural programming. Classes, and dynamic memory allocation are not supported. (That being said, COBOL has over 50 dialects, some more capable than others)
Abstract Data Types	Structs and traits support abstract data types	Lack of native support, data division is used for organizing data.

Reflections

Rust Insights:

Strengths: Expressive concurrency, excellent abstraction, memory safety and contemporary syntax.
Areas for Improvement: Enterprise development tools, wider implementation in legacy systems.

COBOL Insights:

Strengths: Well-established community, robust enterprise systems, and a mature ecosystem.
Areas for Improvement: Verbosity in syntax, limited abstraction, manual memory management.

Given More Time or Hindsight

Rust:

- Examine more complex features such as sophisticated trait implementations and lifetimes.
- Investigate Rust's ecosystem for enterprise applications.

COBOL:

- Improve sectioning to maximize code organization and readability.
- Examine outside resources to improve dependence and build management.

Challenges

Rust:

Challenge: Euler problems often involve algorithmic and mathematical challenges. Rust's strong type system sometimes made it challenging to implement certain mathematical operations elegantly.

Solution: Rust's extensive standard library and a focus on performance helped address many algorithmic challenges. However, expressing certain mathematical ideas required careful consideration of type conversions and precision.

COBOL:

Challenge: COBOL is more aligned with business applications, and expressing complex mathematical operations, especially those found in some Euler problems, was less straightforward.

Solution: COBOL's focus on simplicity and readability led to breaking down complex mathematical operations into more procedural steps. However, this increased verbosity and made the code less elegant compared to Rust.

Einstein Challenges:

COBOL compiler has a file name character limit, this is because a file name, and a subprogram name are the same thing to the compiler. You don't link anything, you write "CALL filename..." inside the program and that parameter name has a built in character limit, so some script names are not in the euler-N-task-name.cbl format. Additionally, the COBOL compiler creates source files by default, not executables so a specific flag is needed. Furthermore, COBOL comes in many dialects, and for the scripts to compile, to make it work putting down a flag for FREEFORM dialect was necessary.

References

"The Rust Programming Language" by Steve Klabnik and Carol Nichols: This book provides an in-depth understanding of Rust's features, including ownership, borrowing, and lifetimes.

"COBOL Programming" by D. Ghosh Dastidar and M. K. Roy: COBOL programming principles, such as divisions, sections, and data division, are covered in this book.

"The Art of Multiprocessor Programming" by Nir Shavit and Maurice Herlihy: This book delves into the ideas and practices of concurrent programming, offering insights on parallelism and synchronization that are pertinent to COBOL's procedural approach to concurrent task handling and Rust's support for concurrency.

Appendix

Problem 1:

To find the sum of all natural numbers below 10 that are multiples of either 3 or 5, we consider the numbers 3, 5, 6, and 9, resulting in a sum of 23. Now, extend this concept to find the sum of all multiples of 3 or 5 below 1000.

Problem 4:

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is 9009, which is the result of multiplying 91 by 99. Now, the task is to find the largest palindrome made from the product of two 3-digit numbers.

Problem 3:

The prime factors of 13195 are 5, 7, 13, and 29.
Now, the task is to find the largest prime factor of the number 600851475143.

Problem 10:

The sum of primes below 10 is calculated as 2 + 3 + 5 + 7, resulting in a sum of 17. Now, the objective is to find the sum of all primes below two million.

Problem 100:

If a box contains twenty-one colored discs, consisting of fifteen blue discs and six red discs, and two discs are taken at random, the probability of drawing two blue discs, denoted as P(BB), is calculated as (15/21) × (14/20) = 1/2. The next such arrangement, where there is exactly a 50% chance of drawing two blue discs at random, involves a box with eighty-five blue discs and thirty-five red discs. To find the first arrangement with over 1,000,000,000,000 discs in total, determine the number of blue discs that the box would contain.

Problem 78:

Let $p(n)$ represent the number of different ways in which n coins can be separated into piles. For instance, five coins can be arranged in seven different ways, so (5)=7 $p(5)=7$. The task is to find the smallest value of n for which $p(n)$ is divisible by one million.

Problem 114:

A row, seven units in length, accommodates red blocks with a minimum length of three units. These red blocks, which can be of different lengths, are positioned such that any two red blocks are separated by at least one grey square. There are precisely seventeen ways of achieving this configuration. The question is: How many ways can a row, fifty units in length, be filled? Please note that while the provided example may not showcase it, in general, it is allowed to mix block sizes. For instance, on a row measuring eight units in length, you could use red (3), grey (1), and red (4).

Problem 117:

By combining grey square tiles and oblong tiles selected from red tiles (measuring two units), green tiles (measuring three units), and blue tiles (measuring four units), it is possible to tile a row, five units in length, in exactly fifteen different ways. The task is to determine how many ways a row measuring fifty units in length can be tiled.

Problem 160:

For any N, let $f(N)$ be the last five digits before the trailing zeroes in $N!$. For example:

- $9!=362880$, so $f(9)=36288$.
- $10!=3628800$, so $f(10)=36288$.
- $20!=2432902008176640000$, $f(20)=17664$.

The goal is to find $f(1{,}000{,}000{,}000{,}000)$

Problem 162:

In the hexadecimal number system, numbers are represented using 16 different digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

For instance, the hexadecimal number AF, when written in the decimal number system, equals $10\times16+15=175$

In the 3-digit hexadecimal numbers 10A, 1A0, A10, and A01, the digits 0, 1, and A are all present. Similar to numbers written in base ten, we write hexadecimal numbers without leading zeroes.

The question is: How many hexadecimal numbers containing at most sixteen hexadecimal digits exist with all of the digits 0, 1, and A present at least once? Provide your answer as a hexadecimal number.

Please ensure the answer is in uppercase (A, B, C, D, E, and F), without any leading or trailing code that marks the number as hexadecimal, and without leading zeroes (e.g., 1A3F and not 1a3f, not 0x1a3f, not $1A3F, not #1A3F, and not 0000001A3F)