Towards a Transpiler for C/C++ to Safer Rust

Dhiren Tripuramallu
IIT Bhubaneswar, India
dtm11@iitbbs.ac.in

Swapnil Singh
IIT Bhubaneswar, India
ss93@iitbbs.ac.in

Shrirang Deshmukh
IIT Bhubaneswar, India
dsp13@iitbbs.ac.in

Srinivas Pinisetty
IIT Bhubaneswar, India
spinisetty@iitbbs.ac.in

Shinde Arjun Shivaji
Samsung R&D Inst., Bangalore, India
shinde.arjun@samsung.com

Raja Balusamy
Samsung R&D Inst., Bangalore, India
raja.balu@samsung.com

Ajaganna Bandeppa
Samsung R&D Inst., Bangalore, India
ajju.b@samsung.com

Abstract

Rust is a multi-paradigm programming language developed by Mozilla that focuses on performance and safety. Rust code is arguably known best for its speed and memory safety, a property essential while developing embedded systems. Thus, it becomes one of the alternatives when developing operating systems for embedded devices. How to convert an existing C++ code base to Rust is also gaining greater attention.

In this work, we focus on the process of transpiling C++ code to a Rust codebase in a robust and safe manner. The manual transpilation process is carried out to understand the different constructs of the Rust language and how they correspond to C++ constructs. Based on the learning from the manual transpilation, a transpilation table is created to aid in future transpilation efforts and to develop an automated transpiler. We also studied the existing automated transpilers and identified the problems and inefficiencies they involved. The results of the transpilation process were closely monitored and evaluated, showing improved memory safety without compromising performance and reliability of the resulting codebase. The study concludes with a comprehensive analysis of the findings, an evaluation of the implications for future research, and recommendations for the same in this area.

Keywords: Rust, C/C++, source-to-source compiler

1 Introduction

Rust [3, 13] is a systems programming language that provides developers with low-level access and control over system resources such as memory, file systems, and networking. This makes it ideal for building software applications that require performance, efficiency, and reliability, such as databases, compilers, and other low-level systems [8, 13].

Rust's notable feature is its absence of a garbage collector, distinguishing it from most other high-level languages. Rust achieves memory safety through its sophisticated type system that tracks variable lifetimes at compile-time. This enables Rust to automatically insert optimized LLVM/assembly instructions for memory deallocation, resulting in improved performance and stability [3]. In addition to its low-level control and performance, the automatic memory allocation and deallocation also makes Rust a memory-safe language, which prevents common programming errors such as null pointer dereference and buffer overflows.

Why migrate to Rust?: Given these features, it is easy to see why Rust is an attractive option for organizations that need to migrate their existing infrastructure to a new platform. By using Rust, organizations can take advantage of its low-level control, performance, and memory safety to build software that is faster, more efficient, and less prone to errors.

Rust also has a strong focus on safety and security. The language enforces strict ownership and borrowing rules, making it difficult for developers to write code that can result in undefined behaviour, race conditions, and other security issues. This helps organizations reduce the risk of security vulnerabilities in their software and improve the overall reliability of their systems. Another important aspect of Rust is its highly optimized runtime system. The Rust compiler is designed to generate highly efficient machine code, making it a good choice for performance-critical systems. Rust also provides a number of performance-related features such as inline functions and zero-cost abstractions, which can help developers write high-performance code [4] that is both readable and maintainable. Given the range of benefits that Rust offers, it is an excellent choice that can help an organization achieve its goals and succeed in a rapidly changing technological landscape.

Problem and Contributions: Organizations such as *Samsung Electronics* are interested in migrating their existing C++ codebases to Rust without starting from scratch. Therefore, a reliable auto transpiler for C++ to Rust is crucial. In our study, we focused on the *Tizen OS*, which is utilized by Samsung Electronics in their smart devices. We extensively explored existing auto transpilers and evaluated their ability to generate safe Rust code, using both simple programs and

```
1   fn main() {
2       let v = vec![1, 2, 3];
3       let v2 = v;
4       println!("v[0] is: {}", v[0]);
5       /* error:use of moved value:`v`
6       println!("v[0] is:{}", v[0]);*/
7   }
```

Figure 1. Example of ownership in Rust

a full-scale Tizen's gperf module. Our analysis included assessing the accuracy and reliability of the generated code, identifying strengths and weaknesses of each tool. Unfortunately, our findings revealed that the existing transpilers were inconsistent in producing reliable and safe Rust code.

Based on the study of the existing auto transpilers, there is a need to explore further on building a safe and reliable transpiler for C++ to Rust, that led to the following main contributions of this work.

In this work, the focus is on understanding the different constructs of Rust and how they correspond to the constructs of C++. A deep dive into the language and its features is conducted, including the correlation between these constructs and how they can be implemented in Rust. For example, we explored how Rust's concept of references and borrowing replaces C++'s raw pointers and null pointers.

We then proceeded to manually transpile a module, the gperf module of Tizen platform (originally written in C++), as a proof of concept. This manual process helped us gain a deeper understanding of the language and its constructs. Based on the learning from this process, we have created a transpilation table. This transpilation table that resulted will guide future transpilation efforts and also form the basis for the creation of an automated transpiler. We then also compared and provided some insights on the performance of C++ vs. Rust on the gperf module.

2   Some key concepts of Rust

Rust is a programming language that is designed to prioritize memory safety. To achieve this, Rust provides several features that help prevent common programming errors that can lead to crashes and security vulnerabilities. Here are some of the fundamental but important to the domain memory safety features in Rust. Please refer to Rust's official website [5] for complete documentation on the features.

```
1   fn main () {
2       fn take(v: Vec<i32>) {
3           // Implementation
4       }
5       let v = vec![1, 2, 3];
6       take(v);
7       println!("v[0] is: {}", v[0]);
8       /* error: use of moved value: `v`
9       println!("v[0] is: {}", v[0]);*/
10  }
11
```

Figure 2. Example of ownership using functions in Rust

Ownership. Rust's ownership system ensures that there is always exactly one binding to any given resource. When we assign a value to another binding, we are actually transferring ownership of that value. This transfer is known as a move in Rust, and it helps to prevent common bugs such as use-after-free errors and data races.

In Figure 1, a new vector *v* is created and initialized with the values 1, 2, and 3. Then a new variable *v2* is created and assigned *v*. This means that ownership of the data in *v* is transferred to *v2* and *v* can no longer be used. An attempt is made to access the first element of *v* after it has been moved to *v2*, but this results in a compile-time error because *v* has been moved and is no longer available in the current scope.

The same error occurs in Figure 2 when the ownership of *v* is passed to function ***take***. The function takes ownership, and the original variable can no longer be used.

Borrowing. It is allowed in Rust to borrow a value, which grants temporary access to the variable. Instead of taking variables as arguments in functions, references of the variables are taken as arguments. This way, ownership of the resource is borrowed rather than owned. There are 2 types of borrowing both important for the domain, namely Mutable Borrows, and Immutable Borrows.

Rules of Borrowing: In Rust, borrowing rules state that a borrow must not outlast its owner's scope, and only one mutable reference or one or more references to a resource can exist at a time. These rules prevent data races by ensuring that multiple pointers cannot access the same memory location at the same time, with at least one of them writing, without synchronization.

We discuss further about the concept of borrowing and the rules of borrowing in Rust via some examples in *Appendix B* available in the repository [2].

3   Related Work

Rust, released in 2015, has consistently been named the "most loved programming language" for seven years in a row according to the Stack Overflow Developer Survey [1]. Its popularity has resulted in numerous research papers exploring its applications in software development and programming languages. The active Rust developer community also contributes to the abundance of online discussions, blog posts, and tutorials.

Studies have focused on migrating from languages like C/C++ to Rust due to its advantages in speed and safety, particularly for developing low-level systems. Open-source migration guides, such as "A Guide to Porting C/C++ to Rust," are available to assist developers in this process.

Martins [12] investigated the benefits and drawbacks of using Rust in an existing codebase, specifically the EPICS framework. The author found that if EPICS had been written

in Rust, approximately 41 bugs could have been easily identified and fixed. The study recommended selectively rewriting critical components in Rust, while maintaining a well-defined interface between the two languages.

Various automatic transpilation tools are available for converting C/C++ code to Rust, including C2Rust [7], CRust [17] and bindgen [14]. These tools offer different levels of functionality and limitations, which will be discussed in the subsequent sections of this paper.

Another study by Ling et al. [10] proposed a transpiler that transforms C code into safer Rust code by reducing the usage of the "unsafe" keyword. The authors' evaluation on open-source and commercial C projects demonstrated significantly higher ratios of safe code after the transformations. Similarly, Lunnikivi et al. [11] discussed transpiling Python code to Rust to improve performance while maintaining readability. They proposed using Rust as an intermediate step for code optimization. The authors presented a transpilation process involving optional runtime types, the use of the pyrs [9] tool, manual refactoring, and validation testing. Their approach showed performance gains compared to accelerated Python implementations.

In the next section, we will explore the effectiveness of the aforementioned transpilation tools, providing a comprehensive analysis of their features and limitations.

4 C/C++ to Rust: On the existing tools

In this section, we focus on evaluating the effectiveness of existing tools for transpiling C/C++ code to Rust. We begin by exploring the available tools and selecting a set of representative programs to test their transpilation capabilities. We then compare the resulting Rust code with the original C/C++ code to evaluate the quality of the transpilation. Through this evaluation, we aim to provide insights into the current state of C/C++ to Rust transpilation tools and their potential for adoption in real-world applications.

4.1 Examples considered and approach

To evaluate the performance of automated C/C++ to Rust transpilers, we selected various sets of C/C++ code that include common constructs in both languages. The goal was to analyze how well the transpiler can convert C/C++ code into memory-safe Rust code that performs the same task. The description of codes chosen for evaluation are given below and the exact codes are included in *Analysis_Examples* available in the repository [2].

4.1.1 C Language.

- **Recursive Fibonacci Numbers**: This program (shown in *Fibonacci.c* available in the repository [2]) was chosen to study how Rust code handles function calls, including self-calls and calls to other functions, when transpiled.
- **Linked List Implementation in C**: This program (shown in *LinkedList.c* available in the repository [2]) utilizes structs and raw pointers, which have a distinct implementation in Rust. However, raw pointers are not encouraged in safe

Rust code. Transpiling this program aids in comprehending how a transpiler converts structs and raw pointers.

4.1.2 C++ Language.

- **Catalan Numbers Problem**: This is a basic C++ code that uses simple constructs like loops (shown in *CatalanNumbers.cpp* available in the repository [2]). The conversion of this program shows how these constructs are dealt with during the transpilation process.
- **Basic Classes and Object Code**: This program (shown in *BasicOOPS.cpp* available in the repository [2]) is a basic implementation of Classes and Objects, i.e. OOP in C++. OOP is an essential part of Tizen development, and this program was selected to understand how classes and objects are converted during the transpilation process.

4.2 Tools Explored

This Subsection describes the automated transpilation tools we have identified and explored. Each tool has an overview along with its issues. The full code transpilation results for each tool are available in *Analysis_Results* available in the repository [2].

4.2.1 C2Rust. C2Rust [7] is an open-source tool for converting C code to Rust code. It aims to automate the process of porting C code, reducing the time and effort required for manual translation while preserving the original behaviour and performance of the C code. However, there are some limitations to consider when using C2Rust.

C2Rust is a powerful tool, but has some limitations.

- **Complex C code**: C2Rust may struggle with complex C code that uses advanced language features, macros, or low-level system calls. The tool is imperfect and may generate incorrect or inefficient Rust code for such cases.
- **Dependencies**: C2Rust can only convert the C code itself, not any external dependencies or libraries. These dependencies must be ported manually or replaced with existing Rust libraries.
- **Performance**: The converted code may not perform as well as the original C code, especially if the C code is highly optimized. It is important to profile the converted code and optimize performance where necessary.
- **Manual intervention**: The conversion process with C2Rust is not perfect and may require manual intervention to correct errors or improve code quality. The converted code should be thoroughly tested and reviewed to ensure it behaves as expected.
- **Rust-specific concepts**: C2Rust may not always be able to translate complex C code into Rust code that follows best practices for the Rust programming language. The generated code may require manual modification to adhere to Rust conventions and idioms.
- **C99 support only**: C2Rust currently only supports C99 code, and there is no support for C++. This means that if

```
1    /** Crust doesn't resolve C/C++
     dependencies or included header.
2    * You may have to define your own
     module and implement those
     functionality in Rust
3    * Or you can translate header file
     with Crust to produce Rust code. *
4    * >>>>>>>> # include < bits / stdc
     ++ . h >
5    **/
```
6   **Figure 3.** Code excerpt from the transpilation

the original code is written in a different dialect of C or in C++, it may not be able to be converted with C2Rust.

- **Unsafe code**: Since C2Rust primarily focuses on automating the conversion process, the generated Rust code may contain unsafe elements. This is because many of the unsafe constructs in Rust are used to match the behaviour of the C code. As a result, manual review and modification may be necessary to make the generated code safer and more idiomatic Rust.

C2Rust is a tool designed to convert C code into Rust. Although it can be a valuable asset for porting C code, there are some important limitations to keep in mind. Firstly, C2Rust currently only supports C and does not have the capability to handle C++ code. Secondly, the converted code in Rust is unsafe, meaning that the user does not get the advantage of memory safety and increased speed by directly using the code. To use these benefits, the user needs to put in significant effort, roughly 90-95%, into converting the output code to safe Rust. This requires a thorough understanding of Rust programming and a strong attention to detail to ensure that the code is safe and secure. Despite these limitations, C2Rust can still be a useful tool for those looking to port their C code to Rust, but it should be used with caution and with a deep understanding of the consequences of converting unsafe code.

4.2.2 bindgen. Rust bindgen [14] is a tool that automates the process of creating Foreign Function Interface (FFI) bindings between Rust and C/C++ libraries. It allows developers to access C libraries directly in their Rust code by creating bindings that match the functions and data structures of the C code. Thus, it is important to note that Rust bindgen is not a transpiler; it generates FFI bindings for C/C++ code. It creates bindings that enable Rust to interact with the C/C++ code but does not convert the code into Rust. Therefore, the safety of these bindings relies on the memory safety of the original C/C++ code, and Rust bindgen does not add any additional safety measures.

As a result, the transpilation of C/C++ code into Rust was not pursued in this context. The objective was to fully transpile the modules into Rust rather than simply relying on the C library. This approach ensures the codebase is fully converted and can take advantage of Rust's safety and performance benefits.

```
1    // Importing Token Crate
2    use crate::library::lexeme::token::
     Token;
```
3   **Figure 4.** Crate import example

4.2.3 CRust. CRust is an open-source project on GitHub created by a developer based in Bangalore [17]. It is designed as a typical compiler and involves lexical analysis and parsing of the input source code. Although CRust has been able to solve several issues faced by previous tools, it still has some limitations that need to be considered. The limitations of CRust include:

- **Header file conversion**: CRust is unable to convert included header files and resolve dependencies or header files in C/C++. As seen in Figure 3 it provides message stating the same in the output file whenever it finds a header file in the source code. A workaround is to convert the header files separately and then include them using the Rust crate syntax as shown in Figure 4.
- **Unknown corresponding functions**: In cases where the transpiler does not know the corresponding functions in Rust, it directly copies the function from the C/C++ file. This will require manual correction or the use of a C library package. C Library functions *printf* and *getchar* in the source code of Figure 5 are copied as it is in Figure 6, their Rust counterparts are not present in the transpiled code.
- **Class support**: CRust claims to support classes in C++, but its performance is not optimal. It may throw errors for index out of bounds, enter into an infinite loop during execution, or have other issues with its core logic for parsing class declarations.
- **Preprocessors**: There is code that supports preprocessors like HeaderDefine, HeaderInclude, HeaderIfDefineStart, and HeaderIfDefineEnd, but it is currently not in a working state. This can be extended by changing the code.

4.2.4 Verdict on tools. The study explored the following tools for automatic transpilation of C/C++ code to Rust.

```
1    int main(){
2      int n = 9;
3      printf("%d\n", fib(n));
4      getchar();
5      return 0; }
6
```
    Figure 5. Example C Code

```
1    fn main() {
2      let n: i32 = 9;
3      printf("%d\n", fib(n));
4      getchar();
5      return 0; }
6
```
    Figure 6. Resultant Rust Code

- **C2Rust** - This tool supports only C, not C++, and the converted code is in unsafe Rust, which requires almost 90-95% effort to be converted to safe Rust.
- **bindgen** - This tool creates a Foreign Function Interface to access the C/C++ code. It is not a transpiler and not suitable for the purpose of transpilation.
- **CRust** - This transpiler is suitable for basic C/C++ code but requires manual intervention for conversion to Rust. Class conversion is not supported, which limits its use case.

We concluded that CRust seemed to be the most promising tool for converting small chunks of code and can be used for about 40% of the transpilation work of basic code. However, if the issue of class transpilation can be fixed, it may provide a head-start in transpilation. Also, some tools (such as Corrode [16]) could not be built and are no longer supported. Therefore, those tools were not evaluated.

4.3 Transpilation of *gperf* module using CRust

To assess the feasibility of migrating Tizen modules to Rust, we attempted to transpile existing source files in the Tizen's *gperf* module using the CRust tool. As previously stated, **CRust** is the only automatic transpilation tool that can be considered for converting C/C++ to safe Rust code. We compared the outcome of the automatic transpilation with that of the manual transpilation. The comparison was based on the percentage of lines of C/C++ code that were converted to equivalent Rust code. The goal was to determine the accuracy of the automatic transpilation performed by CRust and to evaluate how much manual intervention would be required.

Conversion Results: The *Conversion results table* available in our repository [2] presents the conversion results of each file in the gperf module. The full transpilation of the files is included in *transpiled_gperf* available in the repository [2].

The table shows that while some files could not be transpiled, others had conversion percentages ranging from 5% to 45%, with logic fragments of the code being successfully converted. However, some data types were not added to the variables, and some functions were ignored or copied as they are. Thus, it suggests that CRust can partially convert C/C++ code to Rust, but additional effort will be required to complete the conversion process.

4.4 Summary- existing transpilation tools

The CRust transpiler's performance fell short of expectations. It struggled with larger code sections, failed to convert header files with class definitions into appropriate Rust equivalents. Furthermore, complications arose with .icc files, necessitating their conversion to .cpp format for analysis. The transpiler encountered difficulties in handling certain crucial source code files, leading to infinite loops. While it managed to convert smaller code snippets resembling C/C++ syntax, such as array access and arithmetic operations, some code was either disregarded or mistaken for comments due to issues with the comment delimiter. On a positive note, variable data types were accurately assigned in Rust. Overall, the CRust transpiler's outcome was disappointing, as much of the code was merely copied with transpiler comments signaling the need for further exploration of corresponding Rust translations.

Upon assessing the current state of automated transpilers, it becomes apparent that they cannot satisfactorily convert Tizen modules from C/C++ to Rust. The Tizen codebase heavily relies on Object-Oriented Programming (OOP) principles and extensively employs memory pointers and inline functions. The primary obstacle lies in the disparities between memory and class handling in C/C++ and Rust. Additionally, differences in syntax compound the challenges encountered during the transpilation process. These combined factors render existing transpilers ill-suited for successful conversion of Tizen modules.

5 Mapping of C/C++ to Rust

We began by taking the gperf module of Tizen platform into consideration for manually transpiling the C++ files present inside it into the Rust codebase. The following are some of the fundamental, key and significant constructs of C++ with their equivalent implementations in Rust. And it's very important to note that the ideal transpilation would essentially be from any form (safe/unsafe) of C++ code to safe Rust. This includes memory-safe and unsafe C++ code.

5.1 Basic Constructs

Let us see how some basic constructs in C++ such as global variables, and ternary operator are implemented in Rust.

5.1.1 Global Variables. The Rust language offers two ways, using const and static, and keywords.

```
1    const const_global: f32 = 2.4;
2    static static_global: i32 = 10;
3    static mut mut_global: i32 = 5;
4    fn main() {
5      unsafe{
6        mut_global = mut_global + 1;
7      }
8    }
```

Figure 7. Global Variables in Rust

- **const/static without *mut* keyword**: To declare immutable global variables.
- **static with *mut* keyword**: To declare mutable global variables. However, in order to access the *static mut* variable, we need to wrap it with the unsafe code.

To avoid the use of **unsafe{}** code, it is better to use types from *std::sync* module such as *RwLock* or *Mutex* that provide thread-safe ways to mutate shared state without the need for the unsafe keyword.The *let* keyword is not permitted to be used in the global scope.

The code snippet in Figure 7 defines an immutable constant ***const_global*** of type ***f32*** with an initial value of ***2.4***, an immutable static variable ***static_global*** of type ***i32*** with an initial value of ***10*** and a mutable static variable ***mut_global***

of type *i32* with an initial value of *5*. *mut_global* is accessed in an *unsafe* block.

```
1        let a = if x > 5 {10} else {7};
```
Figure 8. Ternary Operator Equivalent in Rust

5.1.2 Ternary Operator. Unlike other languages, Rust does not have ternary operators. It uses the regular if-else construct. The code in Figure 8 is an example of a conditional expression in the Rust programming language. The variable *a* is assigned a value based on the result of the condition $x > 5$. If *x* is greater than *5*, *a* is assigned the value *10*, otherwise it is assigned the value *7*. This is a concise way of writing a basic if-else statement.

5.1.3 Vector. A contiguous growable array type: Vec⟨T⟩.

Using Vec::new() method. The Rust code in Figure 9 demonstrates two ways of creating a vector (a dynamic array) of signed 32-bit integers. The first method creates an empty vector using *Vec::new()* and then inserts the integer *1* into it using *vec.push(1)*. The second creates an empty vector with a pre-allocated *capacity of 5* using *Vec::with_capacity(5)*. This can be useful if the program knows the expected size of the vector in advance to avoid costly reallocations as elements are added to the vector later on.

```
1    let mut vec:Vec<i32>=Vec::new();
2    vec.push(1);
3    //Construct empty vector with
     certain capacity
4    let mut vec: Vec<i32> = Vec::
     with_capacity(5);
5
```

Figure 9. Vector in Rust using new() method

```
1        let v = vec![0, 2, 4, 6];
2        println!("{}", v.len());
3        //Prints the number of
     elements in the vector
4        //Loop to iterate over the
     vector
5        for i in v {
6           //iterating through i on the
     vector
7           print!("{} ",i);   }
8
```

Figure 10. Vector in Rust using vec! macro

Using vec! macro –. The Rust code in Figure 10 defines a vector *v* containing the values *[0, 2, 4, 6]*. The first block of code prints the number of elements in the vector using the *len()* method of the vector. The second block of code uses a for loop to iterate over the elements of the vector. The loop variable *i* takes on the value of each element in the vector *v* in turn, allowing the loop body to perform some operation on each element. In this case, the loop body simply prints each element separated by a space.

5.1.4 Do-While Loops. We can implement it using a regular loop block with a condition to break at the end. The Rust code in Figure 11 demonstrates a loop construct, that is an infinite loop, that repeatedly calls a function *doStuff()* as long as a condition, *c* is met.

```
1    loop {
2        doStuff();
3        if !c { break; }
4    }
```

Figure 11. Loop in Rust

5.2 Non-Trivial Constructs

In this section, we show how some of the fundamental, yet non-trivial constructs, such as functions, classes and pointers in C++ are handled in Rust along with emphasizing on some key concepts in them.

5.2.1 Functions.
 - Functions in C/C++ can be split into their declarations and their implementations as shown in Figure 12.

```
1    // Declaration
2    int foo(bool parameter1, const std
     ::string &parameter2);
3    // Implementation
4    int foo(bool parameter1, const std
     ::string &parameter2) {
5        return 1; }
6
```

Figure 12. Function Declaration and Definition in C++

However, Rust does not allow such distinctions, so both of them has to be put in the same place as shown in Figure 13.

The Rust code in Figure 13 defines a function foo that takes two parameters: a boolean value *parameter1* and a reference to a string slice *parameter2*, and returns an integer value of *1*.
 - Rust also does not support function overloading. So to implement such traits in Rust, we can declare functions with different names and their respective parameter configurations as shown in Figure 14.

```
1    fn foo(parameter1: bool, parameter2:
       &str)->i32 {
2    // implementation
3    1 }
4
```

Figure 13. Function in Rust

```
1    fn functAdd3(a: i32, b: i32, c: i32)
       ->i32{
2    // implementation
3    a+b+c }
4    fn functAdd2(a: i32, b: i32) -> i32{
5    //implementation
6    a+b }
7
```

Figure 14. Function Overloading Alternative in Rust

The Rust code in Figure 14 defines two functions *functAdd3* and *functAdd2* that perform addition of integer values. The function *functAdd3* takes three parameters *a*, *b*, and *c*, all of type *i32* (signed 32-bit integer) and returns the sum of the three numbers. In case of function *functAdd2*, the implementation is similar to *functAdd3*, but takes only two parameters and returns the sum of *a* and *b*.

- Inline functions in C++ as shown in Figure 15 can be implemented in Rust using the #[*inline*] keyword as shown in Figure 16.

```
1   inline void sort_char_set (
    unsigned int *base, int len){
2     //Implementation
3   }
```
Figure 15. Inline Function in C++

```
1   #[inline]
2   fn sort_char_set(base: *mut u32,
    len: i32) {
3     //Implementation
4   }
```
Figure 16. Inline Function in Rust

The Rust code in Figure 16 defines a function *sort_char_set* that sorts an array of unsigned 32-bit integers in place. The *#[inline]* attribute is a hint to the compiler to optimize the function by inlining it at the call site, which can improve performance by reducing the overhead of function calls.

5.2.2 Classes.

- Implementation of class in C++ is shown in Figure 17. Classes in Rust can be implemented using *structs*. A sep-

```
1       Class C{
2         int a;   int b;   int c;
3         public:
4         C(int x, int y, int z)
5         { //constructor
6           a = x;   b = y;   c = z;
7         }
8         void method1(int x, int y)
9         { a = x;   b = y; }
10        int method2()
11        { a = 0;   return c; }
12      }
13    //creating an object
14    C object(x,y,z);
```
Figure 17. Class in C++

arate *impl* block is required in order to specify the struct methods. The above class can be implemented in Rust as shown in Figure 18.

The code in Figure 18 defines a struct *C* with three fields *a*, *b*, and *c*, which are of type *i32*. The *pub* keyword is used to make these fields public, which means they can be accessed from outside the struct. Two member functions *method1* and *method2* are defined for *C* inside the

impl block. *method1* takes two integer parameters *x* and *y*, and updates the values of *a* and *b* using the self reference. *method2* sets the value of *a* to *0* and returns the value of *c*. The *&mut self* reference is used to indicate that *method1* and *method2* modify the fields of the struct, and they take ownership of the struct while they execute. A new object of type *C* is created using the struct initialization syntax *C {a:x, b:y, c:z}*, where *x*, *y*, and *z* are the values of the constructor parameters.

```
1   pub struct C {
2     a : i32,  b : i32,  c : i32,
3     }
4   impl C {
5     pub fn method1(&mut self, x:i32, y:
    i32)
6     {
7       self.a = x; self.b = y;
8     }
9     pub fn method2(&mut self) -> i32
10    {
11      self.a = 0;   self.c
12    }
13  }
14  //constructor creating object
15  let object = C {a:x, b:y, c:z};
```
Figure 18. Struct in Rust

- Important thing to note here is that Rust does not support inheritance. Consider the following simple case of inheritance in C++ as shown in Figure 19.

```
1       struct St1
2       {
3         int a;   int b;
4       }
5       struct St2 : St1
6       {   int c;   }
7
```
Figure 19. Inheritance in C++

Now to implement the same relation between two structs in Rust, we can either add a member of the parent struct, or we can simply add all the members of the parent struct and reinitialize them with the same values as of the parent. The code in Figure 20 defines two structs, *St1* and *St2*, in Rust. *St1* contains two fields of type *i32*, named *a* and *b*. This struct has no parent or derived classes and it stands alone. *St2* also contains two fields of type *i32* named *a* and *b*, and an additional field *c* of type *i32*. The *c* field is specific to *St2* and has nothing to do with *St1*.

Alternatively, the second *St2* struct definition shows how to define a struct with a parent using composition. It has a field named *parent* of type *St1*, which represents the parent struct. The *St2* struct also has a unique field named *c* of type *i32*. By using composition, *St2* inherits the fields of

```
1    struct St1
2    {
3      a : i32,   b : i32,
4    }
5    struct St2
6    {
7      a : i32,  b : i32,   c : i32,
8    }
9    //or
10   struct St2
11   {
12     parent : St1,   c : i32,
13   }
```
Figure 20. Inheritance Alternatives in Rust

St1 through the **parent** field. Also, this **parent** field is not visible outside of the **St2** struct.

In Rust, inheritance is achieved through composition, where a struct can contain a field of another struct type, and the child struct can access the fields of the parent struct through the parent field.

5.2.3 Pointers and References.

- As shown in Figure 21, a pointer in most languages like C/C++ basically is the reference to the actual memory location of where the data is being stored. The same can be done in Rust as shown in Figure 22.

```
1        int age = 27;
2        int *age_ptr = &age;
```
Figure 21. Pointer in C++

```
1  // This is a reference coerced to a
     const pointer
2  let age: u16 = 27;
3  let age_ptr: *const u16 = &age;
4  // This is a mut reference coerced
     to a mutable pointer
5  let mut total: u32 = 0;
6  let total_ptr:*mut u32= &mut total;
```
Figure 22. Raw Pointers in Rust

The Rust code in Figure 22 demonstrates how to create pointers from references in Rust. The first example shows how to create a pointer from an immutable reference using the ***const** keyword. The reference to **age** is created using the **&** operator, and then it is coerced to a const pointer using the ***const** keyword. The second example shows how to create a pointer from a mutable reference using the ***mut** keyword. The mutable reference to **total** is created using the **&mut** operator, and then it is coerced to a mutable pointer using the ***mut** keyword.

The *const keyword is used to specify that the pointer is immutable and cannot be used to mutate the data it points to. The *mut keyword is used to specify that the pointer is mutable and can be used to mutate the data it points to.

Note that most of the functions that we might want to use pointers in would be unsafe by definition. They must be inside an *unsafe* block. Therefore, it is not recommended to use raw pointers in Rust. In Rust, a reference is also lifetime tracked by the compiler.

```
1      float *ptr = new float(10.25);
```
Figure 23. Pointer in C++

- In Rust, Box⟨T⟩ is a smart pointer that can be used to allocate things on the heap similar to *new* in C++. Basically, it is a type that provides ownership and lifetime management for heap-allocated values, and automatically deallocates the memory when it goes out of scope. The following pointer in C++ as shown in Figure 23 can be implemented using Box pointers in Rust as shown in Figure 24.

```
1      let ptr:Box<f32> = Box::new(10.25);
```
Figure 24. Box Pointer in Rust

The Rust code in Figure 24 creates a **Box** smart pointer that points to a heap-allocated **f32** value initialized to **10.25**. In this case, the Box pointer owns the **f32** value and can be moved, but not copied, to other variables or functions.

Raw pointers, i.e., *const and *mut can be obtained from Box pointer using Box::into_raw() as shown in Figure 25.

```
1  let box_ptr:Box<int> = Box::new(5);
2  let raw_ptr:*mut i32 = Box::into_raw
     (box_ptr) as *mut i32;
```
Figure 25. Box Pointer to Raw Pointer Conversion in Rust

- References with lifetime specifier; the main aim of lifetimes is to prevent dangling references, which cause a program to reference data other than the data it is intended to reference. Figure 26 shows various ways in which lifetime specifiers are specified to the references.

```
1      &i32        // a reference
2      &'a i32     // a reference with
     an explicit lifetime
3      &'a mut i32 // a mutable
     reference with an explicit lifetime
```
Figure 26. References in Rust

Now, to use references of one struct as a member of another with lifetime specifiers, we can do as shown in Figure 27.

```
1    struct PositionIterator<'a>
2    { _set : &'a Positions, }
3
```
Figure 27. Reference inside Struct in Rust

The Rust code in Figure 27 defines a struct named **PositionIterator** with a generic lifetime **'a**. The **PositionIterator** struct has one field named **_set** that is a reference to an instance of the **Positions** struct. The lifetime specifier indicates that the **_set** reference is tied to the lifetime of the **PositionIterator** struct. This code can be used to create an iterator over the positions in a **Positions** object. The lifetime specifier is used to ensure that the iterator only has access to valid references to the **Positions** object, and does not outlive it. By defining a lifetime on the **PositionIterator** struct, Rust can check that the iterator's lifetime

is valid and make sure that there are no dangling pointers or other memory errors. This helps ensure memory safety and avoid bugs in Rust code.

- NULL Pointers: there is no NULL in safe Rust unlike in C++. However, null pointers can be used in Rust along with raw pointers, which is unsafe to do so, using null()/null_mut() functions in std::ptr.

```
1  use std::ptr;
2  let p: *const i32 = ptr::null();
3  let p_mut: *mut i32=ptr::null_mut();
4  assert!(p.is_null()&&p_mut.is_null());
```

Figure 28. Null Pointers (using raw pointers) in Rust

In Figure 28, a new immutable pointer *p* and a new mutable pointer *p_mut* of types **const i32* and **mut i32* respectively are initialized it to null pointers using the *null()* and *null_mut()* functions respectively from the *ptr* module.

In safe Rust, using enum *Option* is the closest we get to using NULL. We can use the *None* variant of it to represent no-value.

```
1  let recipient: Option<&str> = None;
2  assert!(recipient.is_none());
```
Figure 29. Option Enum in Rust

In Figure 29, a variable *recipient* is defined as an *Option* of a string slice with an initial value of *None*. The assert statement verifies that the *recipient* variable is indeed *None*.

5.3 Summary and some observations: On understanding the mapping of C++ to Rust, and manual transpilation of the gperf module

As can be seen, transpiling from C/C++ to Rust is not direct. Sometimes, it may require a totally different and indirect way of writing a piece of C/C++ code in Rust. Some features that are supported in one language may not be supported in the other, just like NULL pointer and inheritance concepts that are supported in C++ but not in Rust.

Only a few topics are explained above. There are several other complex concepts such as traits, command line parsing, naming conventions, referencing, error handling, memory disposal, and visibility of fields and methods in structs that includes usage of *pub* keyword and nested structs. Error handling using *Result* enum is discussed in *Appendix A* available in the repository [2].

Though there are C/C++ bindings that can be utilized in Rust, it is not advised to do so as they usually are compatible with unsafe Rust.

The gperf module in C++ was structured as separate header (.h), inline definition (.icc), and implementation (.cc) files. However, in Rust, all code is grouped under a single implementation file (.rs), without any distinction between headers and implementations. Therefore, to translate a C++ module to Rust, we created a corresponding .rs file for each set of .h, .icc, and .cc files.

S.No	File Name	Rust(release) (in milliseconds)	C++ (no optimization) (in milliseconds)	C++ (-o1 optimization) (in milliseconds)
1	bool-array	2	6.3	6.5
2	positions	2.5	8.5	1.6
3	keyword	1.1	2.9	1.4
4	keyword_list	0.5	1.1	1.5
5	options	3.7	8.1	7.8
6	hash-table	2	8	2

Table 1. Benchmarking table: Comparison of the runtimes of various files in the gperf module in different modes.

By organizing code into crates and modules, Rust provides a powerful and flexible way to structure code and manage dependencies. This allows developers to write maintainable and scalable code, while also ensuring that code can be reused across projects and shared with others.

gperf is a perfect hash function generator written in C++. It is used to generate the reserved keyword recognizer for lexical analyzers in several compilers and language processing tools [15, 18]. Overall, the gperf module is a complex system that requires careful management of memory and other system resources. The gperf module implementation in C/C++ consists of almost 8500 lines of code [18]. Using the understanding of the various programming constructs, we were able to manually convert the complete C/C++ code of the gperf module to Rust. The transpiled version of the codebase can be found in the refered repository [2]. In the following section, we have briefly compared both the C/C++ and the Rust version of the gperf module implementations.

6 Comparison of C/C++ to Rust (gperf)

We performed a benchmark comparison between the runtime of C++ code and the corresponding Rust code of the files of the gperf module with some basic test cases on a machine with the following configuration:

- Processor: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.80 GHz; Installed RAM: 16.0 GB (15.9 GB usable)
- System type: 64-bit operating system, x64-based processor
- Operating system: Windows 11 Home

The Rust code was tested in debug mode and in release mode, with the release version being faster than the debug version. In contrast, the C++ code was tested with and without optimization flags (-o1 and -o2). The version without optimization flags was slower than Rust's release version. However, the C++ versions with optimization flags were almost similar to Rust's release version, with Rust still outperforming C++ by a few milliseconds in many cases. Table 1 shows the actual runtimes of each version where each runtime value was calculated as an average of runtimes obtained over 10 runs.

One of the drawbacks of using optimization flags in C++ is that it can sometimes lead to unexpected runtime errors and bugs. Additionally, using optimization flags can sometimes lead to longer compile times and larger executable files [6].

The results of this comparison suggest that Rust's performance is on par with C++. Here, the optimization flags in C++ can help achieve similar performance levels as Rust

release compilation, albeit with slightly higher runtime and potential drawbacks. The research highlights the importance of selecting the right programming language for specific use cases, with Rust being a good choice for low-level systems programming, and C++ being suitable for other domains.

Overall, this benchmark comparison demonstrates the performance of Rust and emphasizes the need for continued exploration and development of Rust for low-level systems programming. In the process, it is also observed that there is a need to carefully consider the use of optimization flags in C++ to balance performance gains with potential drawbacks.

7 Discussion on Challenges Faced

Some of the challenges that were faced in this process were:

- Learning the language: Rust is difficult. It has a complex syntax and a steep learning curve. It is designed to uniquely solve some very challenging problems in programming.
- Understanding Rust constructs: Before we could begin the transpilation we had to have a thorough understanding of the different constructs of Rust and how they correspond to C++ constructs, which required significant time and effort.
- Manually transpiling modules: The process of manually transpiling was time-consuming and prone to errors, as it requires a deep understanding of both the source code and the target language.
- Creating a transpilation table: The process of creating a transpilation table based on the learning from manual transpilation was challenging, as it required us to identify and document key differences between C++ and Rust.
- Transpiling certain language-specific constructs:
  - **Header and source files:** In C++, header files contain the declarations of classes, functions, and variables that are defined in source files. However, Rust does not have a direct equivalent of header files. Our approach was to put the declaration and implementation of the class in one place, but this can lead to a large and unwieldy source file. Another approach is to use Rust modules to organize declarations and definitions, but this requires a significant reorganization of the codebase.
  - **Friend class:** In C++, the friend class construct allows a class to grant access to its private members to another class. However, Rust does not have an equivalent construct. Our approach was to use the crate level visibility in Rust to allow a module to access the private members of another module, but this can lead to decreased encapsulation and increased coupling between modules.
  - **Type inference:** C++ allows for implicit type conversions and coercion, which can result in unexpected behavior. Rust, on the other hand, is a strongly typed language that uses type inference to ensure type safety. This can make the transpilation process challenging, as the

types used in the original C++ code may not be immediately obvious, and converting these types to Rust types can be error-prone.
  - **Pointers and references:** C++ makes extensive use of pointers and references, which can lead to memory management issues such as dangling pointers and memory leaks. Rust, on the other hand, uses a borrow-checking system that ensures memory safety at compile-time. Converting pointers and references from C++ to Rust can be challenging, as the semantics of these constructs are different in the two languages.
  - **Exceptions:** C++ has a built-in exception handling mechanism that allows for graceful error handling. Rust, on the other hand, uses a system of Result and Option types to handle errors. Converting exception handling code from C++ to Rust can be challenging, as the two systems have different semantics and error handling mechanisms.

8 Conclusion

In this work, we aimed to transpile C++ code to Rust in a robust and safe way. We began by exploring existing transpilation tools for C++ to Rust conversion. Our analysis of the current state of the existing tools for C++ to Rust showed that they do not provide adequate conversion.

Thus, we began to understand the mapping of various constructs of both the languages. For our study, we have considered a fragment of the Tizen OS (gperf module of the Tizen OS in C++) implemented in C++. By manually transpiling various components of the considered C++ codebase to Rust, we have created a transpilation table/mapping based on the learning from manual transpilation of various C++ source files to Rust. Our learnings of the mappings of various constructs will help towards developing an auto transpiler from C++ to Rust.

In the process, the considered codebase is successfully completely manually transpiled in to Rust. We then performed benchmark comparisons between the runtime of the C++ code (both debug and with optimization flag) and the corresponding Rust code (release mode). We observed that we achieved better memory safety based on Rust's type system without any compromise on the performance.

Note that manually transpiling C++ code to Rust is a time-consuming and laborious process. The conversion requires careful attention to detail, debugging, and testing, which can take significant time and resources. It is often more efficient to write the code from scratch or to use an automated transpiler. Our attempt of manually transpiling the considered codebase is only to carefully understand the mapping between various constructs of both the languages, to study the feasibility of developing an automated transpiler. The next phase of our research will focus on developing an automated transpiler for C++ to Rust.

In conclusion, our research highlights the potential benefits of using Rust for system programming and demonstrates the feasibility of transpiling C++ code to Rust. We believe that with further development, Rust can become a valuable tool for creating high-performance and memory-safe software.

References

[1] 2022. *Stack Overflow Developer Survey 2022.* https://survey.stackoverflow.co/2022

[2] Anonymous Authors. 2023. CPPtoRust/CppToRustWork: The Appendix for "Towards a transpiler for C/C++ to Safer Rust". "https://github.com/CPPtoRust/CppToRustWork". "[Online; Last accessed:September 2023]".

[3] Jim Blandy. 2015. *The Rust Programming Language: Fast, Safe, and Beautiful.* O'Reilly Media, Inc.

[4] William Bugden and Ayman Alahmar. 2022. Rust: The programming language for safety and performance. *arXiv preprint arXiv:2206.05503* (2022).

[5] Rust Foundation. [n. d.]. Rust. "=https://www.rust-lang.org/".

[6] gcc.gnu.org 2023. *Options That Control Optimization.* gcc.gnu.org.

[7] Immunant. 2017. immunant/C2rust: Migrate C code to rust. "https://github.com/immunant/c2rust". "[Online; Last accessed:November 2022]".

[8] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. 2, POPL, Article 66 (dec 2017), 34 pages. https://doi.org/10.1145/3158154

[9] Julian Konchunas. 2015. konchunas/pyrs: Python to rust transpiler. "https://github.com/konchunas/pyrs". "[Online; Last accessed:12 April 2023]".

[10] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R Cordy, and Ahmed E Hassan. 2022. In rust we trust: a transpiler from unsafe C to safer rust. In *ICSE 2022: Companion Proceedings*. 354–355.

[11] Henri Lunnikivi, Kai Jylkkä, and Timo Hämäläinen. 2020. Transpiling Python to Rust for Optimized Performance. In *SAMOS 2020, Greece, July 5–9, 2020, Proceedings*. Springer, 127–138.

[12] BS Martins et al. 2019. Benefits and Drawbacks of Using Rust in an Existing C/C++ Codebase. (2019).

[13] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. *Ada Lett.* 34, 3 (oct 2014), 103–104. https://doi.org/10.1145/2692956.2663188

[14] rust lang. 2012. The 'bindgen' user guide. "https://github.com/rust-lang/rust-bindgen". "[Online; Last accessed: November 2022]".

[15] Douglas Schmidt and Bruno Haible. 2009. *User's Guide to gperf 3.0.4.* https://git.tizen.org/cgit/platform/upstream/gperf/plain/doc/gperf.pdf

[16] Jamey Sharp. 2016. jameysharp/corrode: C to rust translator. "https://github.com/jameysharp/corrode". "[Online; Last accessed: November 2022]".

[17] Nishanth Shetty. 2017. nishanthspshetty/crust: C/C++ to rust transpiler. "https://github.com/NishanthSpShetty/crust/". "[Online; Last accessed: November 2022]".

[18] Tizen. 2014. gperf module. https://git.tizen.org/cgit/platform/upstream/gperf/.