[Samsung PRISM] End Review Report

Grammar Based C/C++ to Rust Transpiler

Team

1. College Professor(s): Dr. Srinivas Pinisetty (spinisetty@iitbbs.ac.in)
2. Students:
    1. Mithun Chandrashekar (20cs01017@iitbbs.ac.in)
    2. Arnav Kumar Behera (20cs01070@iitbbs.ac.in)
    3. Vedanta Mohapatra (20cs02001@iitbbs.ac.in)
3. Department: Computer Science and Engineering

Date: 18 April 2024

RUST | Grammar based C/C++ to Rust Auto Transpiler

SAMSUNG

Problem Statement

- Rust delivers plethora of promises on memory, concurrency safety. More and more organizations are eager to move their code base to Rust.
- But converting huge C++ codebases to rust is challenging and time consuming task which can be automated with the help of state of the art transpiler
- Manually code migration also poses risk of human errors
- Requires developer to be proficient in C/C++ as well as Rust to produce efficient rust code.

Ajaganna Bandeppa
ajju.b@samsung.com
+91 9844992221

Shinde Arjun Shivaji
shinde.arjun@samsung.com
+91 9766993329

Work-let expected duration ~4 months

3
Members

Expectations

- Deterministic Source to Source transpilation using grammar based approach
- Plug and play architecture to support to cope with rust lang development speed

Training/ Pre-requisites

- Good knowledge of Grammar, Compilers
- Well versed with Rust
- Well versed with C/C++

Aug

Kick Off < 1st Month >
- Understanding of grammars, Compilers.
- Transpilation of basic C/C++ constructs
- This iteration will focus on declarations, arithmetic operations, conditionals and loops etc

Sep

Milestone 1 < 2nd Month >
- Advanced C/C++ construts transpilation
- This iteration will focus on transpiling C++ functions and struts

Oct

Milestone 2 < 3rd Month >
- Handling the concepts of ownership and borrowing in the basic constructs
- Applying the same in Advanced constructs

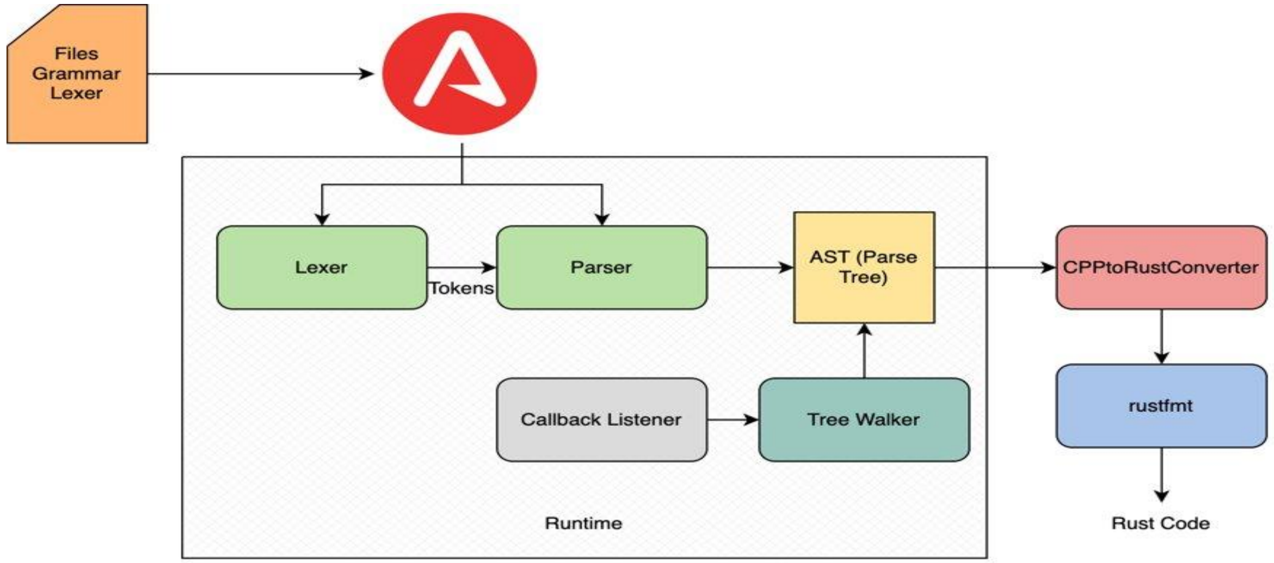
Nov

Closure < 4th Month >
- Testing
- Validation
- Optimization
- Documentation
- Deployment

Approach / Solution

We use a ANTLR generated parser. We derive a parser from the Base parser and override the visitor methods corresponding to non-terminal symbols in the grammar.

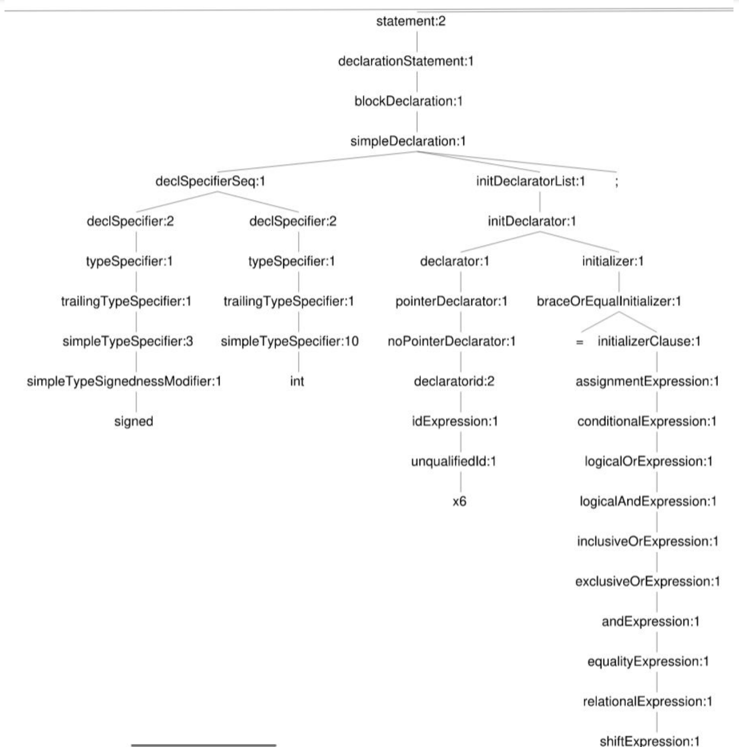
- **Concept Diagram :**



Approach / Solution

The general approach we take is to study the parse tree of C programs and look for the subtrees corresponding to the constructs we wish to transpile and override corresponding visitor methods

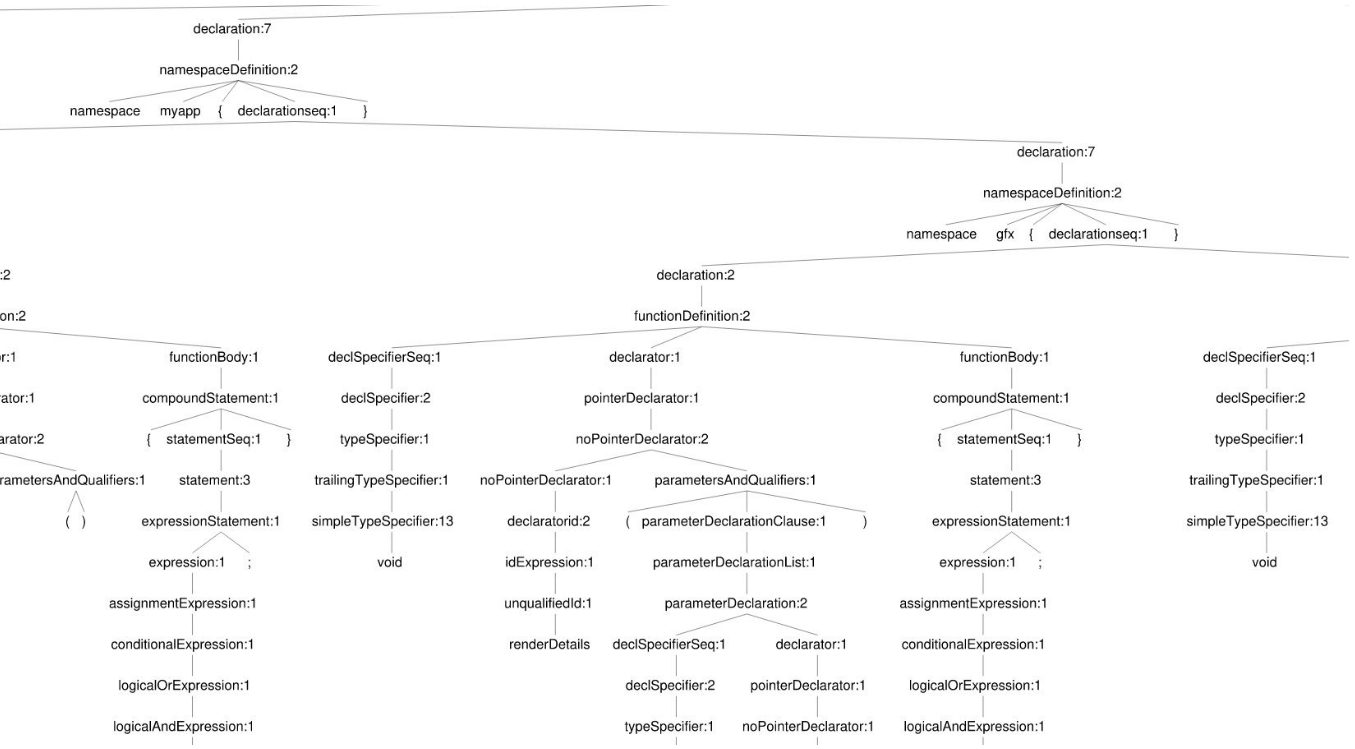
- **A Sample Parse Tree for Simple Declarations** :



- declSpecifierSeq , initDeclaratorList are the relevant non-terminals.
- declSpecifierSeq is used to deal with conversion of types.
- initDeclaratorList is iterated over to get access to multiple declarations in a single line.
- We use both to allow implicit casting.

Approach / Solution

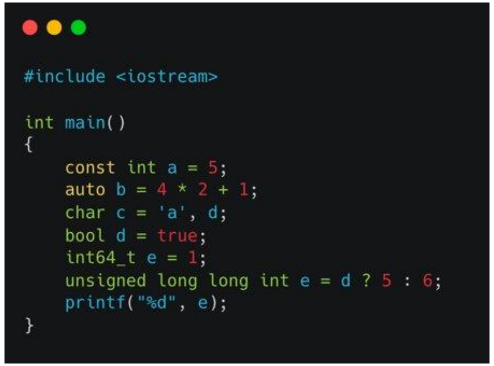
- [A Sample Parse Tree for Namespace Declarations](#) :



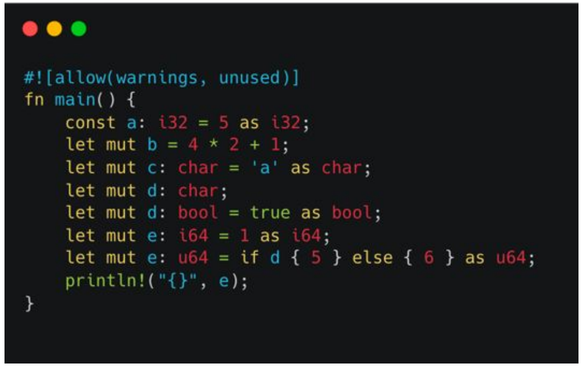
- namespaceDefinition is the relevant nonterminal.
- Namespaces can be nested and therefore we can have them occuring in subtrees of themselves.

Experimental Results

- Standard Declarations (supports more types and length modifiers):

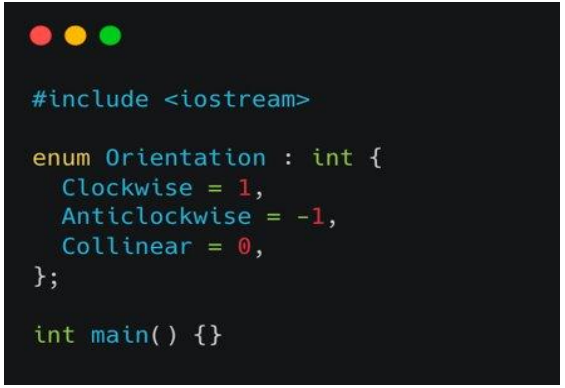


```cpp
#include <iostream>

int main()
{
    const int a = 5;
    auto b = 4 * 2 + 1;
    char c = 'a', d;
    bool d = true;
    int64_t e = 1;
    unsigned long long int e = d ? 5 : 6;
    printf("%d", e);
}
```

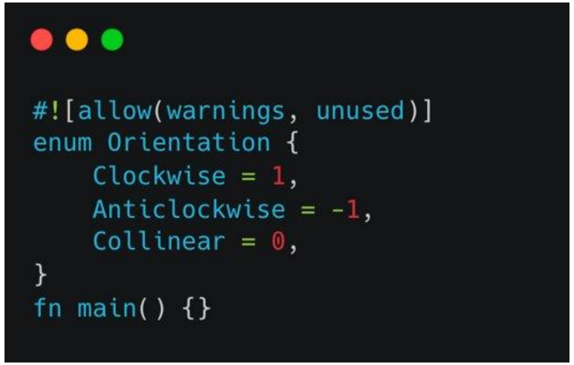


```rust
#![allow(warnings, unused)]
fn main() {
    const a: i32 = 5 as i32;
    let mut b = 4 * 2 + 1;
    let mut c: char = 'a' as char;
    let mut d: char;
    let mut d: bool = true as bool;
    let mut e: i64 = 1 as i64;
    let mut e: u64 = if d { 5 } else { 6 } as u64;
    println!("{}", e);
}
```

- Enumerations :



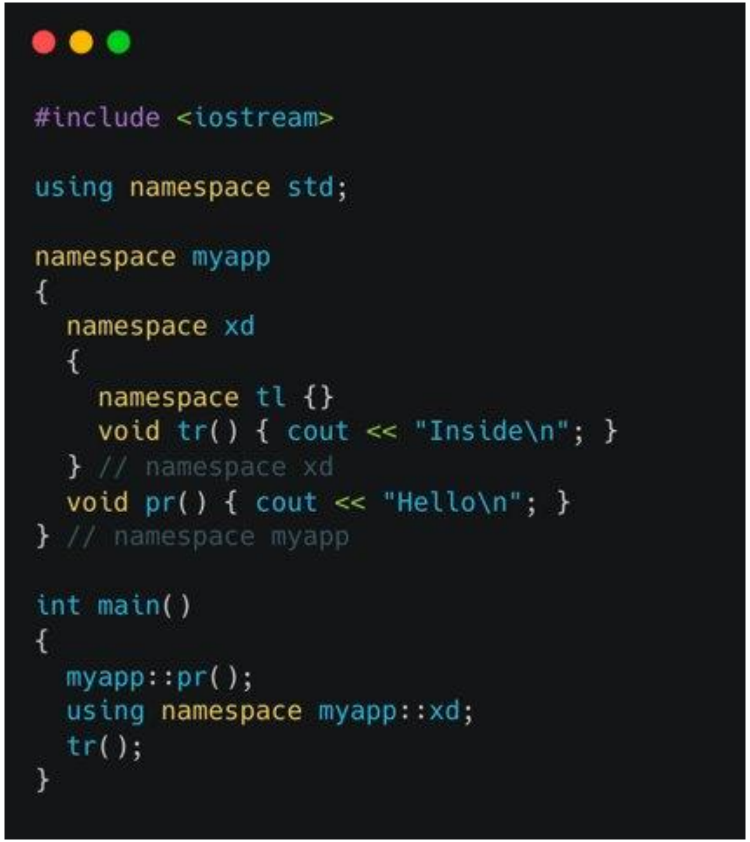
```cpp
#include <iostream>

enum Orientation : int {
  Clockwise = 1,
  Anticlockwise = -1,
  Collinear = 0,
};

int main() {}
```



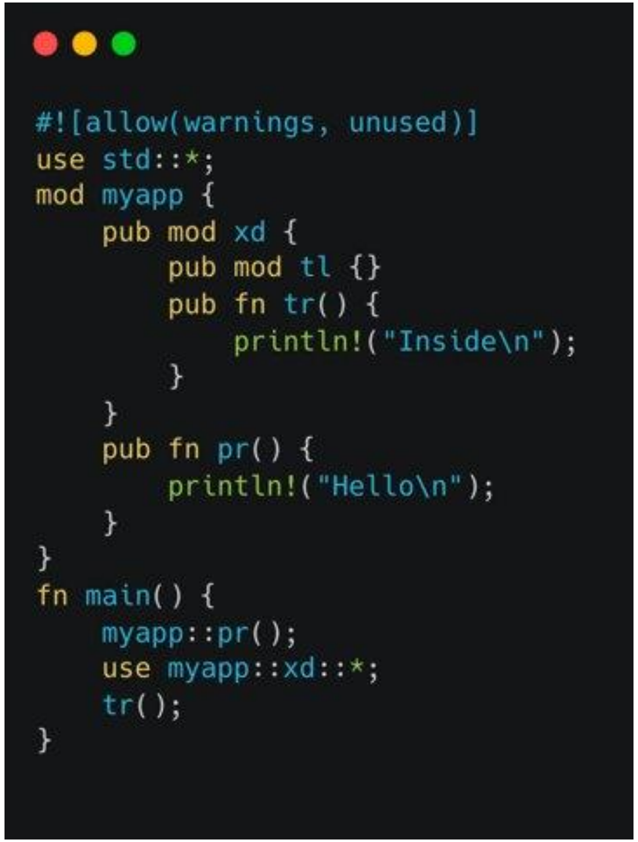
```rust
#![allow(warnings, unused)]
enum Orientation {
    Clockwise = 1,
    Anticlockwise = -1,
    Collinear = 0,
}
fn main() {}
```

Experimental Results

- Namespace Declarations :



```cpp
#include <iostream>

using namespace std;

namespace myapp
{
  namespace xd
  {
    namespace tl {}
    void tr() { cout << "Inside\n"; }
  } // namespace xd
  void pr() { cout << "Hello\n"; }
} // namespace myapp

int main()
{
  myapp::pr();
  using namespace myapp::xd;
  tr();
}
```



```rust
#![allow(warnings, unused)]
use std::*;
mod myapp {
    pub mod xd {
        pub mod tl {}
        pub fn tr() {
            println!("Inside\n");
        }
    }
    pub fn pr() {
        println!("Hello\n");
    }
}
fn main() {
    myapp::pr();
    use myapp::xd::*;
    tr();
}
```

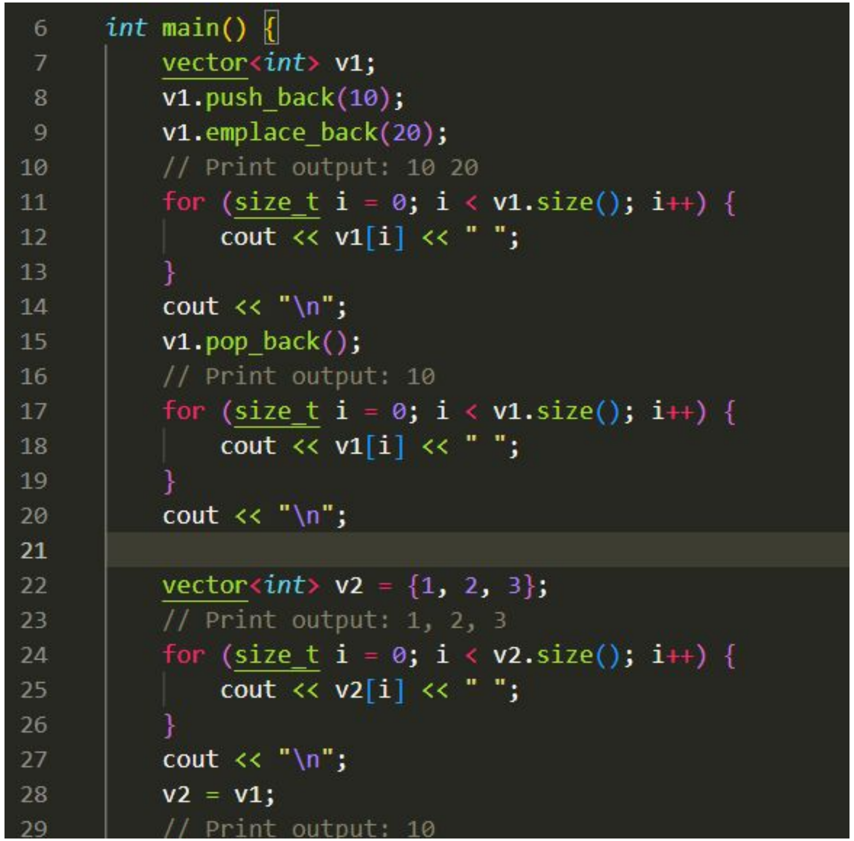
Approach Taken For STL Containers:

- A Sample Parse Tree for Namespace Declarations :

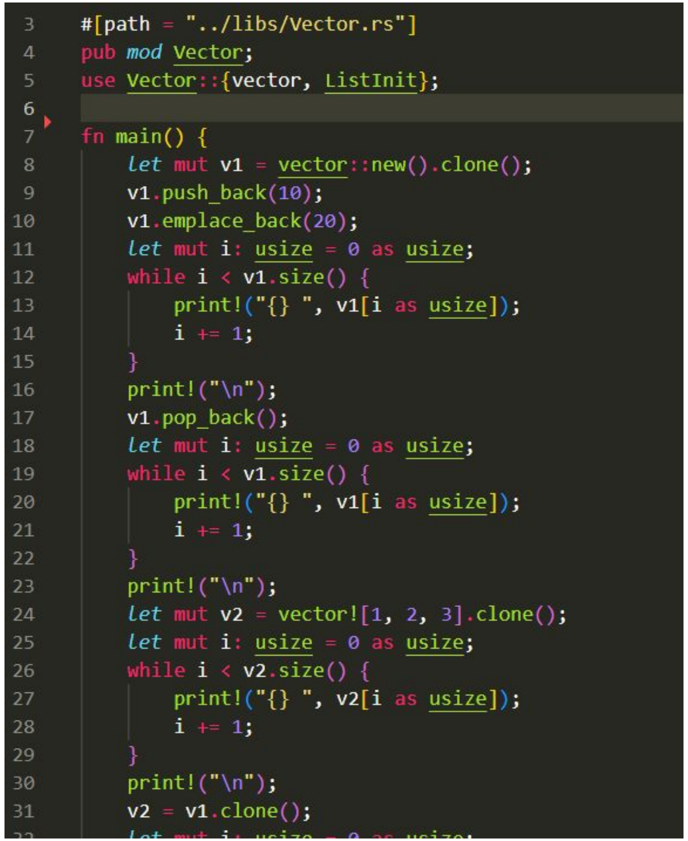
- Instead of transpiling each method call for commonly used containers by mapping corresponding method calls in cpp to rust. (For example for std::vector::push_back() to std::Vec::push())
- We write a struct in a module in rust (that can be imported) that supports the same methods as the cpp counterparts and uses the rust methods internally. This way the method calls can be transpiled as is (though in some cases some modification might be necessary as in swap or operator=)
- The only part requiring significant change would then be the declaration of the objects.

Vectors

- **Sample Programs :**



```cpp
int main() {
    vector<int> v1;
    v1.push_back(10);
    v1.emplace_back(20);
    // Print output: 10 20
    for (size_t i = 0; i < v1.size(); i++) {
        cout << v1[i] << " ";
    }
    cout << "\n";
    v1.pop_back();
    // Print output: 10
    for (size_t i = 0; i < v1.size(); i++) {
        cout << v1[i] << " ";
    }
    cout << "\n";

    vector<int> v2 = {1, 2, 3};
    // Print output: 1, 2, 3
    for (size_t i = 0; i < v2.size(); i++) {
        cout << v2[i] << " ";
    }
    cout << "\n";
    v2 = v1;
    // Print output: 10
```



```rust
#[path = "../libs/Vector.rs"]
pub mod Vector;
use Vector::{vector, ListInit};

fn main() {
    let mut v1 = vector::new().clone();
    v1.push_back(10);
    v1.emplace_back(20);
    let mut i: usize = 0 as usize;
    while i < v1.size() {
        print!("{} ", v1[i as usize]);
        i += 1;
    }
    print!("\n");
    v1.pop_back();
    let mut i: usize = 0 as usize;
    while i < v1.size() {
        print!("{} ", v1[i as usize]);
        i += 1;
    }
    print!("\n");
    let mut v2 = vector![1, 2, 3].clone();
    let mut i: usize = 0 as usize;
    while i < v2.size() {
        print!("{} ", v2[i as usize]);
        i += 1;
    }
    print!("\n");
    v2 = v1.clone();
    let mut i: usize = 0 as usize;
```

Existing Transpilations (Summary):

Construct	Not Supported	Supported
Main function(CLI arguments not supported)		✔
Simple Declarations (fixed size variables and arrays)		✔
Implicit and Explicit casting		✔
Functions		✔
Pointers (Partially)		✔
Structs and classes (partial i.e. lack of abstraction and inheritance)		✔
Exception handling	✔	
NULL values	✔	
Namespaces (Functions but not variables) (Nested Namespaces as well)		✔
Library Functions (Header files for STL Containers)		✔

Existing Transpilations (Summary):

Construct	**Not Supported**	**Supported**
Vectors		✔
Queue / Deque		✔
Stack		✔
Map / Unordered Map		✔
Set / Unordered Set		✔
Strings		✔

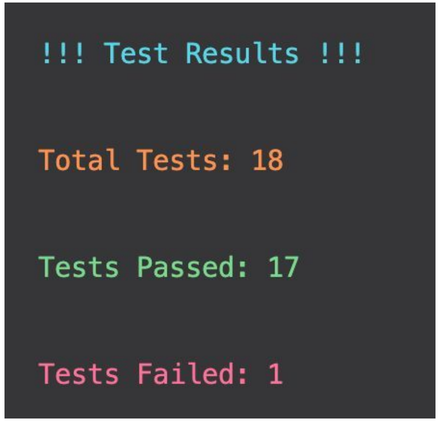
Testing

- Major Observations / Conclusions & Challenges :

For testing our transpiler we use 2 levels of automated testing.

- For Preliminary Testing we compile and run all the transpiled programs in tests/
  This is done using the command `make test`

- Results:



```
!!! Test Results !!!

Total Tests: 18

Tests Passed: 17

Tests Failed: 1
```

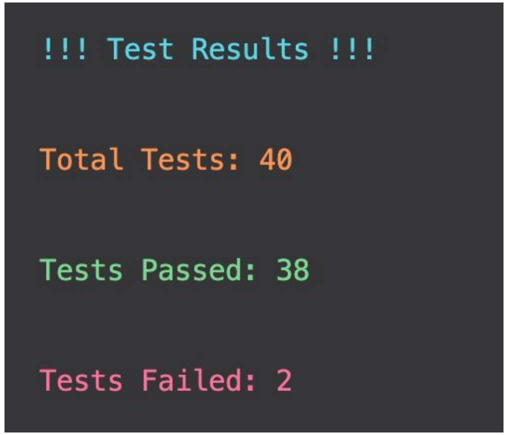
Testing

- Major Observations / Conclusions & Challenges :

For testing our transpiler we use 2 levels of automated testing.

- For more Extensive testing on complex projects. Which have complex dependencies that cannot be resolved by considering one file at a moment, we use the rust formatter to verify the correctness of the translation. This forgoes checking for definitions of functions that may not be present. This is done using the command `make testfmt`

- Results:



```
!!! Test Results !!!

Total Tests: 40

Tests Passed: 38

Tests Failed: 2
```

Observations and Challenges

- **Major Observations / Conclusions & Challenges :**

Challenges Faced:

- Using same visitor function for multiple purposes . For example, modifying visitors corresponding to common non-terminal have long-reaching consequences.
- Transpiling constructs that involve higher level constructs like inheritance.
- No easy one to one mapping available for memory management and garbage collection in rust and C++.

Deliverable

- **Final Deliverables** :
(Discuss in the form of bullets, what are the next steps to complete the solution, any road blocks / bottlenecks, any support needed from SRIB)

- Exception handling, NULL values and inheritance remains an issue which needs to be tackled.
- Conditional compilation were ignored by the grammar file used by the ANTLR parser and so requires changes to the grammar and then additional code to support

- **IP / Paper Publication Plan** :

- A paper can be published which can be an extension to the paper : Towards a Transpiler for C/C++ to Safer Rust (https://arxiv.org/abs/2401.08264).

- **KPIs delivered/Expectations Met**:
(Planned Expectations shared in Work-let vs Delivered Results)

- Transpilation of advanced C/C++ advanced constructs like functions, structs, classes using the grammar based approach.
- Transpilation of commonly used CPP containers namely vectors, sets and maps using adapter classes.

Work-let Closure Details

- **Code Upload details:**

Items	Details
KLOC (Number OF Lines of codes in 000's)	24
Model and Algorithm details	We use a ANTLR generated parser. We derive a parser from the Base parser and override the visitor methods corresponding to non-terminal symbols in the grammar.
Is Mid review, end review report uploaded on Git ?	Yes
Link for Git	https://github.ecodesamsung.com/SRIB-PRISM/IITBHU_23SE09_Grammar_based_C_or_C_plus_plus_to_Rust_Auto_Transpiler

Thank you