[Samsung PRISM] March Review Report

Grammar Based C/C++ to Rust Transpiler

Team

1. College Professor(s): Dr. Srinivas Pinisetty
2. Students:
    1. Mithun Chandrashekar
    2. Arnav Kumar Behera
    3. Vedanta Mohapatra
3. Department: Computer Science and Engineering

Date: 19 Dec 2023

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

Expectations

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

3
Members

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

Pointers:

- Sample Programs

```cpp
#include <iostream>

int main() {
    int p = 1;
    int *q = &p;
    (*q)++;
    (*q) += 1;
    printf("%d", *q);
    printf("%d", p);
}
```

```rust
fn main() {
    let mut p: i32 = 1 as i32;
    let mut // Handling Pointers...
    q: &mut i32 = &mut p as &mut i32;
    (*q) += 1;
    (*q) += 1;
    print!("{}", *q);
    print!("{}", p);
}
```

STL Containers

- Sample Programs

```
5       vector<int> vec;
6       vec.push_back(10);
7       vec.push_back(20);
8       vec.push_back(30);
9       cout << "Element at index 0: " << vec.at(0) << endl;
10      cout << "Element at index 1: " << vec.at(1) << endl;
11      cout << "Element at index 2: " << vec.at(2) << endl;
12      vec.pop_back();
13      cout << "Last element after pop_back(): " << vec.back() << endl;
14      vec.clear();
```

```
2       let mut vec = Vec::<i32>::new();
3       vec.push(10);
4       vec.push(20);
5       vec.push(30);
6       println!("Element at index 0: {}", vec[0]);
7       println!("Element at index 1: {}", vec[1]);
8       println!("Element at index 2: {}", vec[2]);
9       vec.pop();
10      if let Some(last) = vec.last() {
11          println!("Last element after pop(): {}", last);
12      }
13      vec.clear();
```

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

- **Methods Supported:**

operator=	`v2 = v1;`	`v2 = v1.clone();`
swap	`v1.swap(v2);`	`v1.swap(&mut v2);`
push_back	`v1.push_back(10);`	`v1.push_back(10);`
pop_back	`v1.pop_back();`	`v1.pop_back();`
resize	`v2.resize(4, 5);` `v2.resize(4);`	`v2.resize(4, 5);` `v2.resize(4, 0);`
assign	`v2.assign(3, 3);`	`v2.assign(3, 3);`
at	`v2.at(1)`	`v2.at(1)`

Vectors

- **Methods Supported (Contd.) :**

operator[]	`v2[1]`	`v2[1 as usize]`
emplace_back	`v1.emplace_back(20);`	`v1.emplace_back(20);`
front	`v2.front()`	`v2.front()`
back	`v2.back()`	`v2.back()`
empty	`v2.empty()`	`v2.empty()`
size	`v1.size()`	`v1.size()`
declaration 1	`vector<int> v1;`	`let mut v1 = vector::new().clone();`
declaration 2	`vector<int> v2 = {1, 2, 3};`	`let mut v2 = vector![1, 2, 3].clone();`

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();
```

Sets (Ordered and Unordered)

- **Methods Supported:**

operator=	`s2 = s1;`	`s2 = s1.clone();`
swap	`s1.swap(s2);`	`s1.swap(&mut s2);`
insert	`s1.insert(10);`	`s1.insert(10);`
erase	`s1.erase();`	`s1.erase();`
empty	`s1.empty();`	`s1.empty();`
size	`s1.size();`	`s1.size();`
count	`s1.count(10);`	`s1.count(10);`

Vectors

- **Methods Supported (Contd.) :**

declaration 1 (set)	`set<int> s1;`	`let mut s1 = set::new().clone();`
declaration 2 (set	`set<int> s2 = {1, 2, 3};`	`let mut s2 = set![1, 2, 3].clone();`
declaration 1 (unordered set)	`unordered_set<int> s1;`	`let mut s1 = unordered_set::new().clone();`
declaration 2 (unordered set	`unordered_set<int> s2 = {1, 2, 3};`	`let mut s2 = unordered_set![1, 2, 3].clone();`

Set

- **Sample Programs :**

```cpp
#include <iostream>
#include <set>
using namespace std;

int main() {
    set<int> s1, s2 = {1, 2, 3};
    // Insert elements
    s1.insert(1);
    s1.insert(2);
    s2.insert(3);
    // Check if empty and size
    cout << "S1 empty? " << s1.empty() << ", size: " << s1.size() << endl;
    // Count elements
    cout << "Count of 1 in S1: " << s1.count(1) << endl;
    // Erase element
    s1.erase(1);
    cout << "Erase 1, S1 size: " << s1.size() << endl;
    // Swap sets
    s1.swap(s2);
    cout << "After swap, S1 size: " << s1.size() << ", S2 size: " << s2.size() << endl;
    // Clear set
    s1.clear();
    cout << "After clear, S1 size: " << s1.size() << endl;
}
```

```rust
#![allow(warnings, unused)]

#[path = "../libs/Set.rs"]
pub mod Set;
use Set::set;

fn main() {
    let mut s1 = set::new().clone();
    let mut s2 = set![1, 2, 3].clone();
    s1.insert(1);
    s1.insert(2);
    s2.insert(3);
    print!("S1 empty? {}, size: {}\n", s1.empty(), s1.size());
    print!("Count of 1 in S1: {}\n", s1.count(1));
    s1.erase(1);
    print!("Erase 1, S1 size: {}\n", s1.size());
    s1.swap(&mut s2);
    print!(
        "After swap, S1 size: {}, S2 size: {}\n",
        s1.size(),
        s2.size()
    );
    s1.clear();
    print!("After clear, S1 size: {}\n", s1.size());
    return 0;
}
```

Queue/Deque

- **Methods Supported:**

operator=	`q2 = q1;`	`q2 = q1.clone();`
swap	`q1.swap(q2);`	`q1.swap(&mut q2);`
push_back	`q1.push_back(10);`	`q1.push_back(10);`
push_front	`q1.push_front(10);`	`q1.push_front(10);`
pop_back	`q1.pop_back();`	`q1.pop_back();`
pop_front()	`q1.pop_front();`	`q1.pop_front();`
resize	`q2.resize(4, 5);` `q2.resize(4);`	`q2.resize(4, 5);` `q2.resize(4, 0);`

Queue/Deque

- **Methods Supported (Contd.) :**

operator[]	q2[1]	q2[1 as usize]
emplace_back	q1.emplace_back(20);	q1.emplace_back(20);
emplace_front	q1.emplace_front(20);	q1.emplace_front(20);
front	q2.front()	q2.front()
back	q2.back()	q2.back()
empty	q2.empty()	q2.empty()
size	q1.size()	q1.size()
declaration 1	deque<int> q1; queue<int> q1;	let mut q1 = deque::new().clone();
declaration 2	deque<int> q2 = {1, 2, 3}; queue<int> q2 = {1, 2, 3};	let mut q2 = deque![1, 2, 3].clone();

Queue/Deque

- **Methods Supported (Contd.):**

assign	`q2.assign(3, 3);`	`q2.assign(3, 3);`
at	`q2.at(1)`	`q2.at(1)`
push (for queue) (internally push_back)	`q2.push(1)`	`q2.push(1)`
pop (for queue) (internally pop_front)	`q2.pop()`	`q2.pop()`

Queue/Deque

- **Sample Programs :**

```cpp
int main() {
    // Create a deque
    deque<int> deque1;

    // PushBack
    deque1.push_back(1);
    deque1.push_back(2);
    deque1.push_back(3);

    // Display
    cout << "Deque 1: ";
    for (size_t i = 0; i < deque1.size(); ++i) {
        cout << deque1[i] << " ";
    }
    cout << endl;

    // Create another deque
    deque<int> deque2 = {6, 7, 8, 9, 10};

    // PushBack
    deque2.push_back(4);
    deque2.push_back(5);

    cout << "Deque 2: ";
    for (size_t i = 0; i < deque2.size(); ++i) {
        cout << deque2[i] << " ";
    }
    cout << endl;

    // Swap
    deque1.swap(deque2);
```

```rust
#[path = "../libs/Deque.rs"]
pub mod Deque;
use Deque::deque;

fn main() {
    let mut deque1 = deque::new().clone();
    deque1.push_back(1);
    deque1.push_back(2);
    deque1.push_back(3);
    print!("Deque 1: ");
    let mut i: usize = 0 as usize;
    while i < deque1.size() {
        print!("{} ", deque1[i as usize]);
        i += 1;
    }
    print!("\n");
    let mut deque2 = deque![6, 7, 8, 9, 10].clone();
    deque2.push_back(4);
    deque2.push_back(5);
    print!("Deque 2: ");
    let mut i: usize = 0 as usize;
    while i < deque2.size() {
        print!("{} ", deque2[i as usize]);
        i += 1;
    }
    print!("\n");
    deque1.swap(&mut deque2);
```

Stack

- **Methods Supported:**

operator=	`s2 = s1;`	`s2 = s1.clone();`
swap	`s1.swap(s2);`	`s1.swap(&mut q2);`
push	`s1.push(10);`	`s1.push(10);`
pop	`s1.pop();`	`s1.pop();`
resize	`s1.resize(4, 5);` `s1.resize(4);`	`s1.resize(4, 5);` `s1.resize(4, 0);`
at	`s2.at(1)`	`s2.at(1)`

Stack

- **Methods Supported (Contd.) :**

operator[]	`s1[1]`	`s1[1 as usize`
emplace	`s1.emplace(20);`	`s1.emplace(20);`
top	`s2.top()`	`s2.top()`
empty	`s2.empty()`	`s2.empty()`
size	`s1.size()`	`s1.size()`
declaration 1	`queue<int> q1;`	`let mut q1 = queue::new().clone();`
declaration 2	`deque<int> q2 = {1, 2, 3};`	`let mut q2 = deque![1, 2, 3].clone();`

Queue/Deque

- **Methods Supported (Contd.):**

assign	`q2.assign(3, 3);`	`q2.assign(3, 3);`
at	`q2.at(1)`	`q2.at(1)`
push (for queue) (internally push_back)	`q2.push(1)`	`q2.push(1)`
pop (for queue) (internally pop_front)	`q2.pop()`	`q2.pop()`

Stacks

- **Sample Programs :**

```cpp
#include <iostream>
#include <stack>

using namespace std;

int main() {
  stack<int> s1;
  if (s1.empty()) { // Vector is Empty
    cout << "Stack is Empty\n";
  } else {
    cout << "Stack is Non-Empty\n";
  }
  s1.push(10);
  s1.push(20);
  s1.emplace(20);
  cout << "S1.size() = " << s1.size() << "\n";
  while (!s1.empty()) {
    cout << s1.top() << " ";
    s1.pop();
  }
  cout << "\n";

  stack<int> s2;
  s2 = s1;
  s1.push(4);
  cout << "Before swap\n";
  cout << " s1.len() = " << s1.size() << "\n";
  cout << " s2.len() = " << s2.size() << "\n";
  s1.swap(s2);
  cout << "After swap\n";
  cout << " s1.len() = " << s1.size() << "\n";
  cout << " s2.len() = " << s2.size() << "\n";
}
```

```rust
pub mod UnorderedSet;
use std::*;
use UnorderedSet::unordered_set;
fn main() {
    let mut s1 = stack::new().clone();
    if s1.empty() {
        print!("Stack is Empty\n");
    } else {
        print!("Stack is Non-Empty\n");
    }
    s1.push(10);
    s1.push(20);
    s1.emplace(20);
    print!("S1.size() = {}\n", s1.size());
    while !s1.empty() {
        print!("{} ", s1.top());
        s1.pop();
    }
    print!("\n");
    let mut s2 = stack::new().clone();
    s2 = s1.clone();
    s1.push(4);
    print!("Before swap\n");
    print!(" s1.len() = {}\n", s1.size());
    print!(" s2.len() = {}\n", s2.size());
    s1.swap(&mut s2);
    print!("After swap\n");
    print!(" s1.len() = {}\n", s1.size());
    print!(" s2.len() = {}\n", s2.size());
}
```

Maps (Ordered and Unordered)

at	`mp.at(1);`	`mp.at(1);`
empty	`mp.empty()`	`mp.empty()`
size	`mp.size();`	`mp.size();`
max_size	`mp.max_size();`	`mp.max_size();`
clear	`mp.clear();`	`mp.clear();`
erase	`mp.erase(1);`	`mp.erase(1);`
swap	`mp.swap(mp2);`	`mp.swap(&mut mp2);`

Maps (Ordered and Unordered)

count	`mp.count(1);`	`mp.count(1);`
insert	`mp2.insert({3, 3});` `mp2.insert(3, 3);`	`mp2.insert((3, 3));`
emplace	`mp2.emplace(3, 3);`	`mp2.emplace(3, 3);`
operator[]	`mp[1];`	`mp[1];`

Map

- **Sample Programs :**

```cpp
int main() {
  map<int, int> mp;
  map<int, vector<string>> vecmp;
  vector<string> v = {"a"};
  vecmp.insert({1, v});
  if (mp.empty()) {
    cout << "Set is empty" << endl;
  } else {
    cout << "Set is not empty" << endl;
  }
  mp.insert({1, 1});
  mp.insert({2, 2});
  cout << "Size of mp:" << mp.size() << endl;
  cout << "Count of 1 in mp:" << mp.count(1) << endl;
  cout << "Count of 3 in mp:" << mp.count(3) << endl;
  mp.erase(1);
  cout << "Size of mp after removing:" << mp.size() << endl;

  // Swap maps
  map<int, int> mp2;
  mp2.insert({3, 3});
  mp2.insert({5, 5});
  mp.swap(mp2);

  cout << "Size of mp after swapping:" << mp.size() << endl;
  cout << "Size of mp2 after swapping:" << mp2.size() << endl;

  mp.clear();
  cout << "size of mp after clear" << mp.size() << endl;
}
```

```rust
fn main() {
    let mut mp = map::new();
    let mut vecmp = map::new();
    let mut v = vector!["a"].clone();
    vecmp.insert((1, v));
    if mp.empty() {
        print!("Set is empty\n");
    } else {
        print!("Set is not empty\n");
    }
    mp.insert((1, 1));
    mp.insert((2, 2));
    print!("Size of mp:{}\n", mp.size());
    print!("Count of 1 in mp:{}\n", mp.count(1));
    print!("Count of 3 in mp:{}\n", mp.count(3));
    mp.erase(1);
    print!("Size of mp after removing:{}\n", mp.size());
    let mut mp2 = map::new();
    mp2.insert((3, 3));
    mp2.insert((5, 5));
    mp.swap(&mut mp2);
    print!("Size of mp after swapping:{}\n", mp.size());
    print!("Size of mp2 after swapping:{}\n", mp2.size());
    mp.clear();
    print!("size of mp after clear{}\n", mp.size());
}
```

Unordered Map

- **Sample Programs :**

```cpp
int main() {
  unordered_map<int, int> mp;
  unordered_map<int, vector<string>> vecmp;
  vector<string> v = {"a"};
  vecmp.insert({1, v});
  if (mp.empty()) {
    cout << "Set is empty" << endl;
  } else {
    cout << "Set is not empty" << endl;
  }
  mp.insert({1, 1});
  mp.insert({2, 2});
  cout << "Size of mp:" << mp.size() << endl;
  cout << "Count of 1 in mp:" << mp.count(1) << endl;
  cout << "Count of 3 in mp:" << mp.count(3) << endl;
  mp.erase(1);
  cout << "Size of mp after removing:" << mp.size() << endl;

  // Swap maps
  unordered_map<int, int> mp2;
  mp2.insert({3, 3});
  mp2.insert({5, 5});
  mp.swap(mp2);

  cout << "Size of mp after swapping:" << mp.size() << endl;
  cout << "Size of mp2 after swapping:" << mp2.size() << endl;

  mp.clear();
  cout << "size of mp after clear" << mp.size() << endl;
}
```

```rust
fn main() {
    let mut mp = unordered_map::new();
    let mut vecmp = unordered_map::new();
    let mut v = vector!["a"].clone();
    vecmp.insert((1, v));
    if mp.empty() {
        print!("Set is empty\n");
    } else {
        print!("Set is not empty\n");
    }
    mp.insert((1, 1));
    mp.insert((2, 2));
    print!("Size of mp:{}\n", mp.size());
    print!("Count of 1 in mp:{}\n", mp.count(1));
    print!("Count of 3 in mp:{}\n", mp.count(3));
    mp.erase(1);
    print!("Size of mp after removing:{}\n", mp.size());
    let mut mp2 = unordered_map::new();
    mp2.insert((3, 3));
    mp2.insert((5, 5));
    mp.swap(&mut mp2);
    print!("Size of mp after swapping:{}\n", mp.size());
    print!("Size of mp2 after swapping:{}\n", mp2.size());
    mp.clear();
    print!("size of mp after clear{}\n", mp.size());
}
```

Operator Overloading

To convert operator overloading methods, we need to define traits in cpp. Current traits supported are all the arithmetic operations(+,-,*,/,%,^) and index. Issues related to the function parameters exist currently.

```
Complex operator+(Complex const &obj) {
  Complex res;
  res.real = real + obj.real;
  res.imag = imag + obj.imag;
  return res;
}
```

```
impl Add<&Complex> for Complex{
    type Output = Self;
    pub fn  add  ( &mut self,   &  obj :  Complex  )  ->   Complex {
    let mut  res  =   Complex ::new();
     res . real  =  self.real  +  obj . real ;
     res . imag  =  self.imag  +  obj . imag ;
     return  res ;
    }
}
```

Observations and Challenges

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

Observations:

- Similar to how code for STL containers were converted, Other library functions could be written in a similar manner. For example, math.h functions.
- Code involving iterators in C++ has no direct counterpart in rust.
- Implementing operator= for structs can be done by using the clone trait.

Thank you