# Towards Macro-Aware C-to-Rust Transpilation (WIP)

**Robbe DE GREEF[1]**, Attilio DISCEPOLI[1], Esteban AGUILILLA KLEIN[2], Théo ENGELS[3], Dr. Ir. Ken HASSELMANN[3], Prof. Antonio PAOLILLO[1]

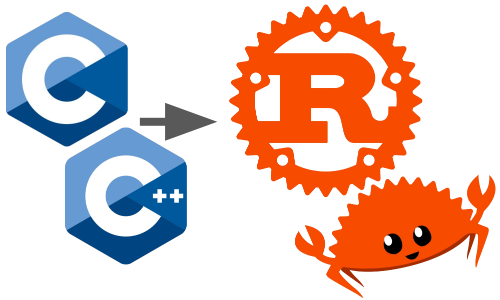
[1]Vrije Universiteit Brussel, [2]Université Libre De Bruxelles, [3]Royal Military Academy of Belgium

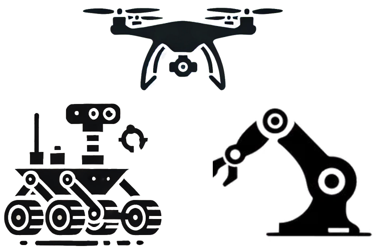
# Context

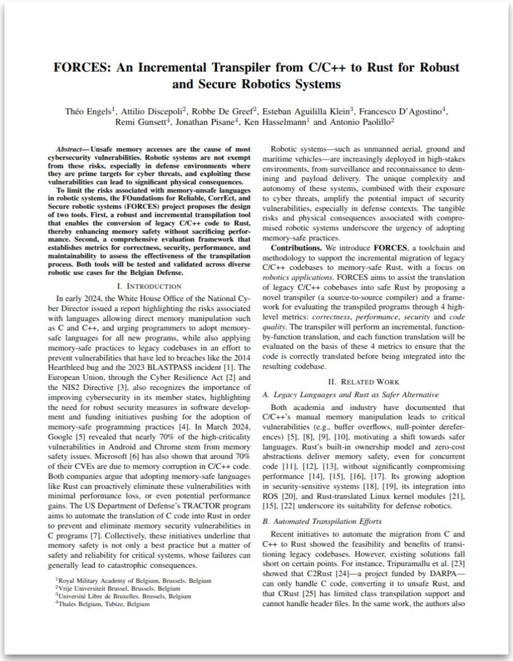
# FORCES project

**Incremental** C/C++ to Rust transpilation project

- Preserving **readability**
- Ensuring **maintainability**







**FORCES: An Incremental Transpiler from C/C++ to Rust for Robust and Secure Robotics Systems**

Théo Engels[1], Attilio Discepoli[2], Robbe De Greef[2], Esteban Aguililla Klein[3], Francesco D'Agostino[4], Remi Gunsett[4], Jonathan Pisane[4], Ken Hasselmann[1] and Antonio Paolillo[2]

*Abstract*— **Unsafe memory accesses are the cause of most cybersecurity vulnerabilities. Robotic systems are not exempt from these risks, especially in defense environments where they are prime targets for cyber threats, and exploiting these vulnerabilities can lead to significant physical consequences.**

**To limit the risks associated with memory-unsafe languages in robotic systems, the FOundations for Reliable, CorrEct, and Secure robotic systems (FORCES) project proposes the design of two tools. First, a robust and incremental transpilation tool that enables the conversion of legacy C/C++ code to Rust, thereby enhancing memory safety without sacrificing performance. Second, a comprehensive evaluation framework that establishes metrics for correctness, security, performance, and maintainability to assess the effectiveness of the transpilation process. Both tools will be tested and validated across diverse robotic use cases for the Belgian Defense.**

I. INTRODUCTION

In early 2024, the White House Office of the National Cyber Director issued a report highlighting the risks associated with languages allowing direct memory manipulation such as C and C++, and urging programmers to adopt memory-safe languages for all new programs, while also applying memory-safe practices to legacy codebases in an effort to prevent vulnerabilities that have led to breaches like the 2014 Heartbleed bug and the 2023 BLASTPASS incident [1]. The European Union, through the Cyber Resilience Act [2] and the NIS2 Directive [3], also recognizes the importance of improving cybersecurity in its member states, highlighting the need for robust security measures in software development and funding initiatives pushing for the adoption of memory-safe programming practices [4]. In March 2024, Google [5] revealed that nearly 70% of the high-criticality vulnerabilities in Android and Chrome stem from memory safety issues. Microsoft [6] has also shown that around 70% of their CVEs are due to memory corruption in C/C++ code. Both companies argue that adopting memory-safe languages like Rust can proactively eliminate these vulnerabilities with minimal performance loss, or even potential performance gains. The US Department of Defense's TRACTOR program aims to automate the translation of C code into Rust in order to prevent and eliminate memory security vulnerabilities in C programs [7]. Collectively, these initiatives underline that memory safety is not only a best practice but a matter of safety and reliability for critical systems, whose failures can generally lead to catastrophic consequences.

[1]Royal Military Academy of Belgium, Brussels, Belgium
[2]Vrije Universiteit Brussel, Brussels, Belgium
[3]Université Libre de Bruxelles, Brussels, Belgium
[4]Thales Belgium, Tubize, Belgium

Robotic systems—such as unmanned aerial, ground and maritime vehicles—are increasingly deployed in high-stakes environments, from surveillance and reconnaissance to demining and payload delivery. The unique complexity and autonomy of these systems, combined with their exposure to cyber threats, amplify the potential impact of security vulnerabilities, especially in defense contexts. The tangible risks and physical consequences associated with compromised robotic systems underscore the urgency of adopting memory-safe practices.

**Contributions.** We introduce **FORCES**, a toolchain and methodology to support the incremental migration of legacy C/C++ codebases to memory-safe Rust, with a focus on *robotics applications*. FORCES aims to assist the translation of legacy C/C++ cobebases into safe Rust by proposing a novel transpiler (a source-to-source compiler) and a framework for evaluating the transpiled programs through 4 high-level metrics: *correctness*, *performance*, *security* and *code quality*. The transpiler will perform an incremental, function-by-function translation, and each function translation will be evaluated on the basis of these 4 metrics to ensure that the code is correctly translated before being integrated into the resulting codebase.

II. RELATED WORK

*A. Legacy Languages and Rust as Safer Alternative*

Both academia and industry have documented that C/C++'s manual memory manipulation leads to critical vulnerabilities (e.g., buffer overflows, null-pointer dereferences) [5], [8], [9], [10], motivating a shift towards safer languages. Rust's built-in ownership model and zero-cost abstractions deliver memory safety, even for concurrent code [11], [12], [13], without significantly compromising performance [14], [15], [16], [17]. Its growing adoption in security-sensitive systems [18], [19], its integration into ROS [20], and Rust-translated Linux kernel modules [21], [15], [22] underscore its suitability for defense robotics.

*B. Automated Transpilation Efforts*

Recent initiatives to automate the migration from C and C++ to Rust showed the feasibility and benefits of transitioning legacy codebases. However, existing solutions fall short on certain points. For instance, Tripuramallu et al. [23] showed that C2Rust [24]—a project funded by DARPA—can only handle C code, converting it to unsafe Rust, and that CRust [25] has limited class transpilation support and cannot handle header files. In the same work, the authors also

https://antonio.paolillo.be/publications/workshops/icra25_r4r_forces_paper.pdf

# No solution exists

| | C2Rust [1] | Corrode [2] | CRust [3] |
|---|---|---|---|
| Convert C to Rust | ✓ | ✓ | ✓ |
| Convert C++ to Rust | ✗ | ✗ | ✓ |
| Generate safe Rust code | ✗ | ✗ | ✗ |
| Preserve comments | ✗ | ✗ | ✓ |
| Preserve macros | ✗ | ✗ | ✗ |
| Generate Rust-like code | ✗ | ✗ | ✗ |
| Actively maintained | ✓ | ✗ | ✗ |

[1] https://c2rust.com [2] https://github.com/jameysharp/corrode [3] https://github.com/NishanthSpShetty/crust

# Problem definition

# C Preprocessor

- **Expansion** occurs **before** compilation
- **Preprocessor** constructs are **not part of C**


Preprocessor must be invoked to produce valid C code

# Why not just expanding them?

```
void skipws(ZFILE *fp) {
 int c;
 while (c=Zgetc(fp),isascii(c)&&isspace(c));
 Zungetc(c,fp);
}
```

→

```
void skipws(ZFILE *fp)
{
 int c;
 while (c = ((--((fp)->len) >= 0) ?
          (unsigned char)(*(fp)->ptr++) :
          _Zgetc(fp)),
         (((c) &
          ~0x7f) == 0) &&
            ((*__ctype_b_loc())[(int)((
               c))] &
            (unsigned short int)_ISspace));
  ((fp)->ptr--, (fp)->len++, (c));
}
```

# Macro contracting

- Turning **expanded macros** back into **invocations**
- **Not always possible**

```c
#define LB {
#define RB }
#define PARENS ()
#define SEMI ;
#define DOUBLE(x) (x * 2)

int main PARENS LB
  return DOUBLE(2) SEMI
RB
```

→

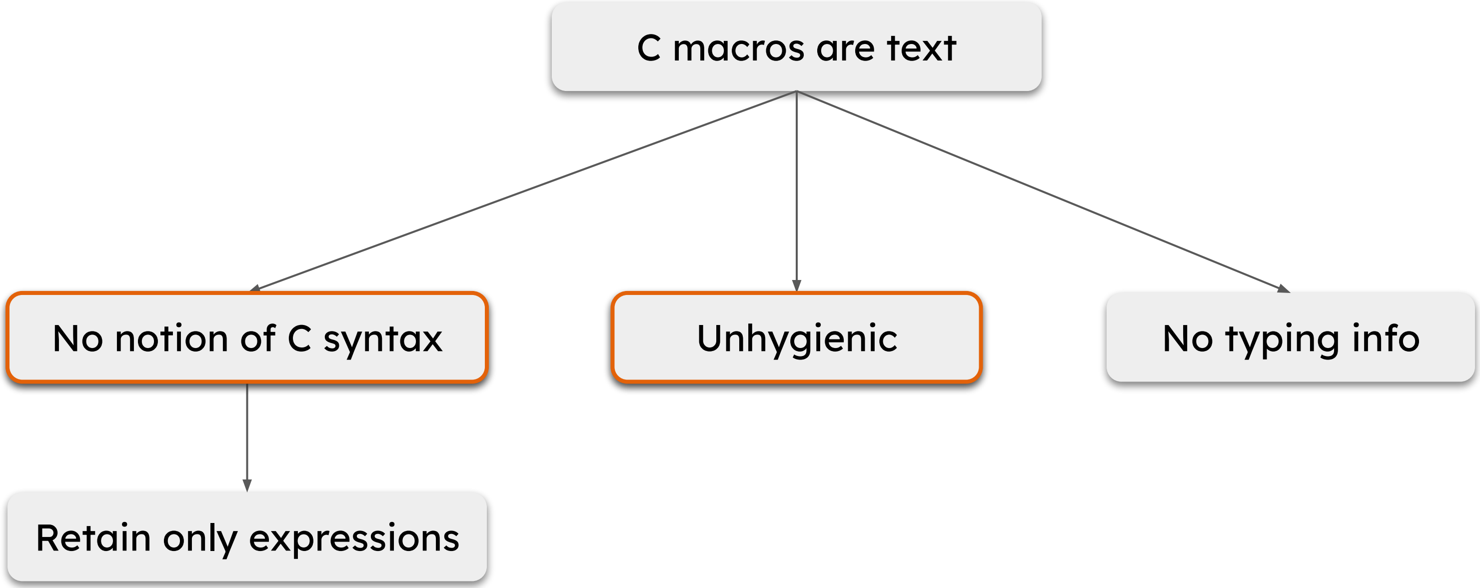
```rust
macro_rules! DOUBLE {
    ($x: expr) => {
        $x * 2
    };
}
```

?

```rust
fn c_main() -> i32 {
    return DOUBLE!(2);
}
```

# Keep expression like macros

- Capture the **majority** of macro **definitions**
  - **~70%** of FIGlet macros
- Generally have an **analogue in Rust**
- **Reduced scope**

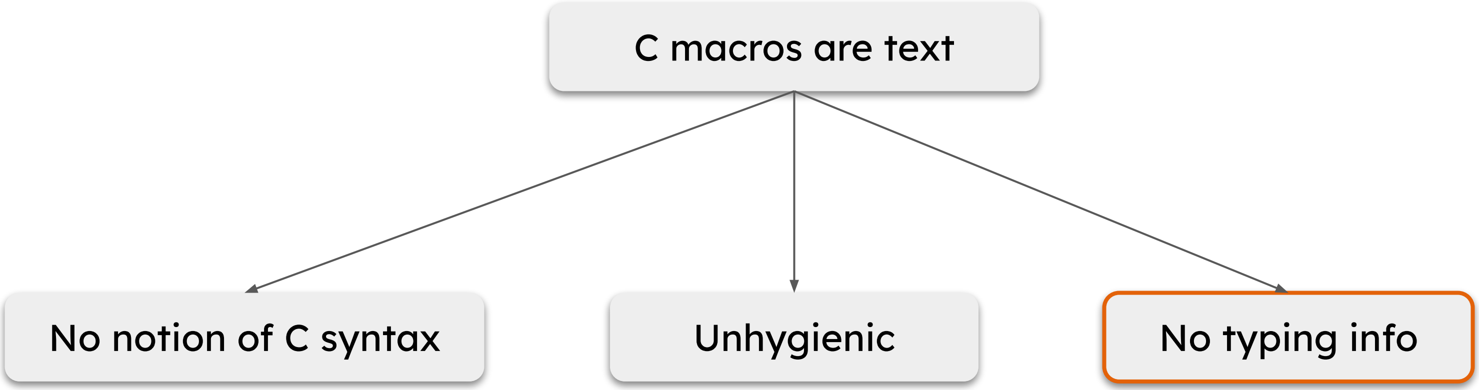

C macros are text
No notion of C syntax
Unhygienic
No typing info
Retain only expressions

# Unhygienic C macros

```
char* strcpy(char*, char*);

int main() {
    int (*strcpy)(char*,char*) = otherfunc;

    char a[] = "Hello";
    char b[] = "abc";

    strcpy((a), (b));
}
```

**Solution**

Check for shadowing of identifiers defined at macro definition


C macros are text
No notion of C syntax
Unhygienic
No typing info

# Intermezzo: implicit type conversions

```c
int main() {
    int a = 2;
    short b = a;
    return b;
}
```

Implicitly →

```c
int main() {
    int a = 2;
    short b = (short) a;
    return b;
}
```

```c
int main() {
    int a = 2;
    short b = 3
    return a + b;
}
```

Implicitly →

```c
int main() {
    int a = 2;
    short b = 3
    return a + (int) b;
}
```

...

C11 (ISO/IEC 9899) — Section 6.3 Conversions

# Intermezzo: implicit type conversions

- Easily solved in regular code

C

```c
int main() {
    int a = 2;
    short b = 3;
    return a + b;
}
```

→

Rust

```rust
fn c_main() -> i32 {
    let a: i32 = 2;
    let b: i16 = 3;
    return a + b as i32;
}
```

matched types

# Arithmetic conversions

```c
#define ADD(x, y) ((x) + (y))

int main() {
    int a = 2;
    short b = 3;

    return ADD(a, b);
}
```

→

Mismatched types

```rust
macro_rules! ADD {
    ($x:expr, $y:expr) => {
        $x + $y
    };
}
fn c_main() -> i32 {
    let a: i32 = 2;
    let b: i16 = 3;
    return ADD!(a, b);
}
```

# Macro typing issues

- Arithmetic conversions
- Pointer arithmetic
- Array to pointer decay
- Assignment conversions
- Integer promotions
- …

Helper traits*

Passing in type parameters



* Different from paper where type parameters were used for arithmetic operations

# Generate helper traits

```
pub trait CAdd<R> {
    type Output;
    unsafe fn c_add(&mut self, rhs: &mut R) -> Self::Output;
}
```

```
impl CAdd::<i16> for i32 {
    type Output = i32;
    unsafe fn c_add(&mut self, rhs: &mut i32) -> i32 {
        ((*self) as i32) + ((*rhs) as i32)
    }
}
```

For all addable types

→

```
macro_rules! ADD {
    ($x:expr, $y:expr) => {
        ($x).c_add($y)
    };
}
fn c_main() -> i32 {
    let a: i32 = 2;
    let b: i16 = 3;
    return ADD!(a, b);
}
```

# Assignment conversions

```
#define ASSIGN(x, y) (x) = (y)

void main() {
    int a = 2;
    short b = 3;
    ASSIGN(a, b);
}
```

→

```
macro_rules! ASSIGN {
    ($x:expr, $y:expr) => {
        $x = $y
    };
}

fn c_main() {
    let mut a: i32 = 2;
    let b: i16 = 3;
    ASSIGN!(a, b);
}
```

Mismatched types

# Pass in types

```c
#define ASSIGN(x, y) (x) = (y)

void main() {
    int a = 2;
    short b = 3;
    ASSIGN(a, b);
}
```

→

```rust
macro_rules! ASSIGN {
    ($x:expr, $y:expr, $ty_0: ty) => {
        $x = $y as $ty_0
    };
}

fn c_main() {
    let mut a: i32 = 2;
    let b: i16 = 3;
    ASSIGN!(a, b, i32);
}
```

# Tying everything together

# Transpiling FIGlet

- ASCII text generator
- All but one file transpiled
- **64/85** macros retained

```
  ___       _     _ _          _ 
 / _ \__  _(_) __| (_)_______ | |
| | | \ \/ / |/ _` | |_  / _ \| |
| |_| |>  <| | (_| | |/ /  __/|_|
 \___//_/\_\_|\__,_|_/___\___|(_)
```

# Example from earlier

```c
void skipws(ZFILE *fp) {
 int c;
 while (c=Zgetc(fp),isascii(c)&&isspace(c));
 Zungetc(c,fp);
}
```

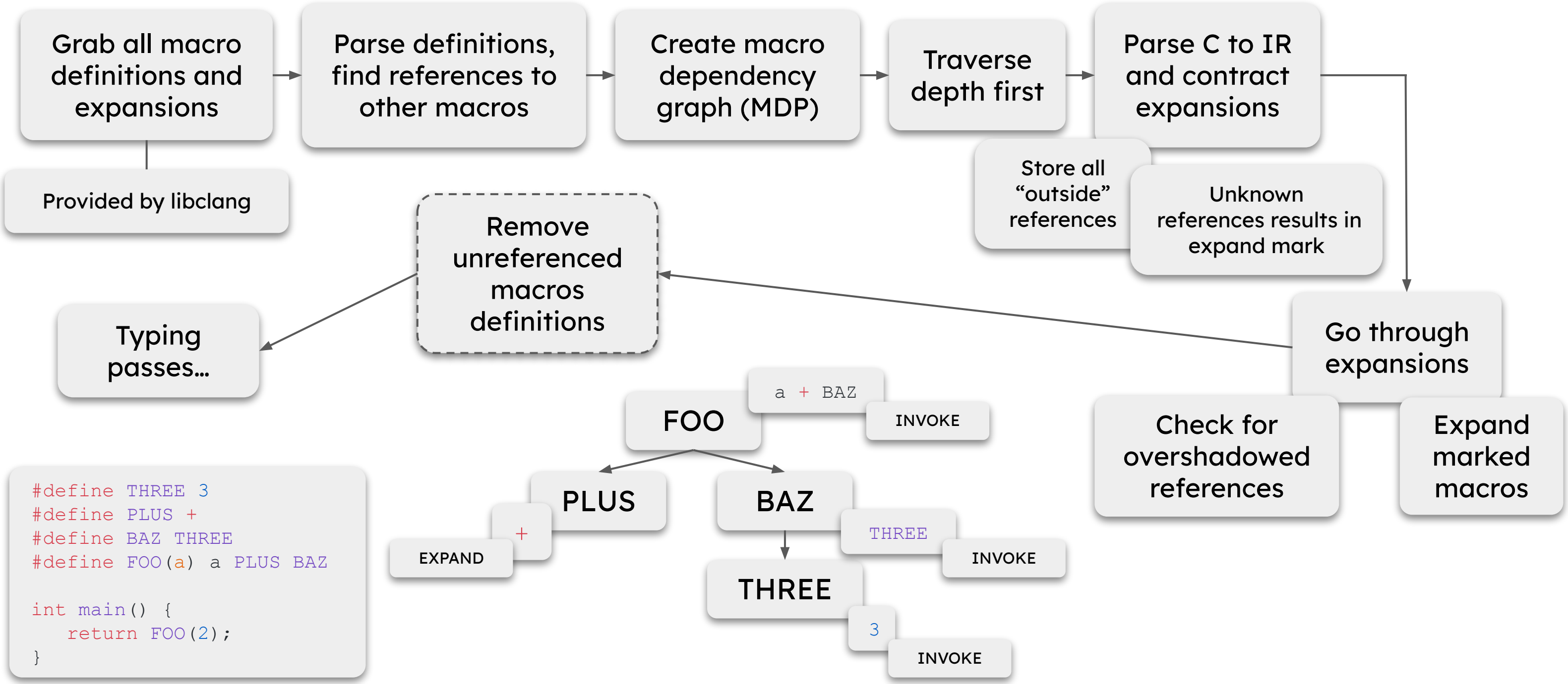
# Example from earlier

```
pub unsafe extern "C" fn skipws(mut fp: *mut ZFILE) -> () {
    let mut c: i32 = (0 as i32);
    while ({
        c = (Zgetc!(fp, *mut ZFILE, i32, i32, *mut ZFILE, *mut u8, *mut ZFILE, i32, i32, u32) as i32);
        (((isascii!(c, i32, i32, i32, i32, i32) != 0)
            && (isspace!(c, *mut *const u16, *const u16, u32, u32) != 0)) as i32)
    } != 0) {}
    Zungetc!(c, fp, *mut ZFILE, *mut ZFILE);
}
```

# Future cleanup

```
pub unsafe extern "C" fn skipws(mut fp: *mut ZFILE) -> () {
    let mut c: i32 = (0 as i32);
    while ({c = (Zgetc!(fp) as i32); (isascii!(c) != 0 && isspace!(c) != 0) as i32} != 0) {}
    Zungetc!(c, fp);
}
```

# There is more…



# Demo

# Backup slides

# C2Rust skipws example

```rust
pub unsafe extern "C" fn skipws(mut fp: *mut ZFILE) {
   let mut c: std::ffi::c_int = 0;
   loop {
       (*fp).len -= 1;
       c = (if (*fp).len >= 0 as std::ffi::c_int {
           let fresh6 = (*fp).ptr;
           (*fp).ptr = ((*fp).ptr).offset(1);
           *fresh6 as std::ffi::c_int
       } else {
           _Zgetc(fp)
       });
       if !(c & !(0x7f as std::ffi::c_int) == 0 as std::ffi::c_int
           && *(*__ctype_b_loc()).offset(c as isize) as std::ffi::c_int
               & _ISspace as std::ffi::c_int as std::ffi::c_ushort as std::ffi::c_int
               != 0)
       {
           break;
       }
   }
   (*fp).ptr = ((*fp).ptr).offset(-1);
   (*fp).ptr;
   (*fp).len += 1;
   (*fp).len;
}
```

# Why no assign trait?

```
trait Foo<T> {
    fn assign(&mut self, other: T) -> ();
}
impl<A, B> Foo<B> for A {
    fn assign(&mut self, other: B) -> () {
        *self = other as A;
    }
}

fn main() {
    let mut x: i32 = 34;
    let y: i16 = 2;
    x.assign(y);
    println!("{x}");
}
```

Non primitive cast

# Why figlet?

- **Small** C program
- Knew it well
- Output **easily verifiable**

# Macro study results

| Filename | Transpiled | Total |
|---|---|---|
| utf8.c | 0 | 0 |
| chkfont.c | 4 | 4 |
| figlet.c | 31 | 31 |
| inflate.c | 10 | 19 |
| zipio.c | 11 | 20 |
| getopt.c | 3 | 4 |
| crc.c | 0 | 1 |

* Numbers differ from paper due to transpiler upgrades

# Macro study DOOM

405/493 macros transpiled (~88%)

# Example (no casting)

```
#define Zungetc(c,f) ((f)->ptr--, (f)->len++, (c))
```

```
macro_rules! Zungetc {($c: expr ,$f: expr) => {{
      {$f.c_deref().ptr.c_post_dec(); $f.c_deref().len. c_post_inc()}; $c}
}}
```

C

```
Zungetc(dummy,fp);
```

Rust

```
Zungetc!(dummy , fp);
```

# One more problem

- Certain casts are not allowed in Rust
- **Non-primitive conversions**

```
fn c_main() {
    let mut x: [i32; 3] = [1, 2, 3];
    let y = x.as_ptr() as *mut i16;
}
```

Non primitive cast

# Broken assign example

```
macro_rules! ASSIGN {
    ($x:expr, $y:expr, $ty_0: ty) => {
        $x = $y as $ty_0
    };
}

fn c_main() {
    let mut a: *mut i16;
    let b: [i32; 3] = [1, 2, 3];
    ASSIGN!(a, b, *mut i16);
}
```

Non primitive cast

# Special cast macro

- Use **specialized helper macro** to cast
- Multiple cast “modes”

```
macro_rules! rust_cast {
    ($lhs: expr, $rhs: ty, regular) => {
        $lhs as $rhs
    };
    ($lhs: expr, $rhs: ty, array_to_pointer) => {
        $lhs.as_ptr() as $rhs
    };
    // ...
}
```

# Fixed assign example

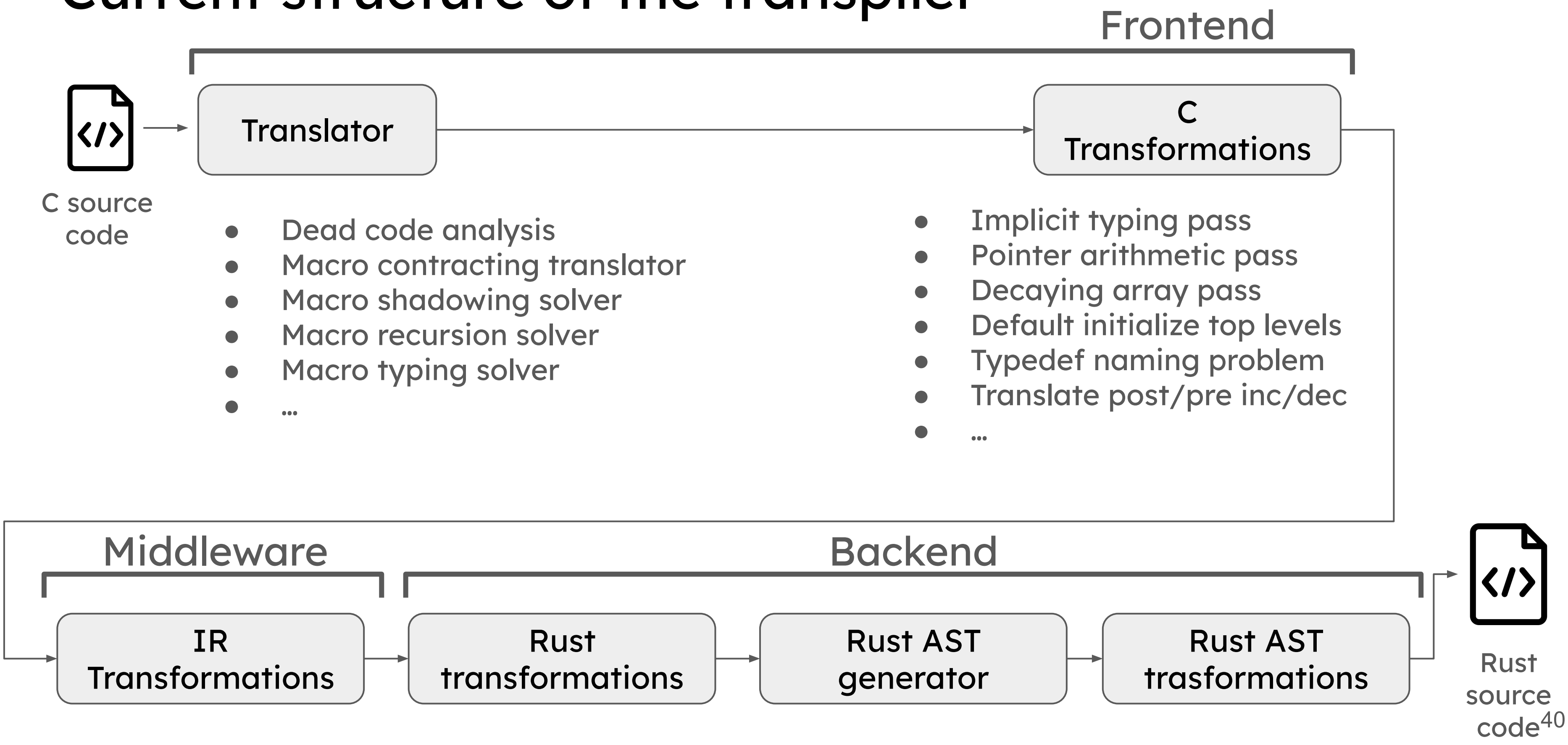
```
macro_rules! ASSIGN {
    ($x:expr, $y:expr, $ty_0: ty, $cast_ty: ident) => {
        $x = rust_cast!($y, $ty_0, $cast_ty)
    };
}

fn c_main() {
    let mut a: *mut i16;
    let b: [i32; 3] = [1, 2, 3];
    ASSIGN!(a, b, *mut i16, array_to_pointer);
}
```

# C Preprocessor

- Uses **preprocessor constructs** to extend C
- **Macros** are one of these constructs

```c
#include <stdio.h>

#define FOO 123
#define PRINT(str) printf(str)

int main() {
    PRINT("Hello, World!\n");
    return FOO;
}
```

# Current structure of the transpiler



# Backup demo video