UCF2021
Rust & OpenSHMEM

Tony Curtis & Rebecca Hassett

Stony Brook University

UCF2021: Rust & OpenSHMEM

1. Tony
   a. Intro to OpenSHMEM and this project
2. Rebecca
   a. Overview of Rust
   b. Rust calling OpenSHMEM

UCF2021: Rust & OpenSHMEM

RustySHMEM Project Motivation

- OpenSHMEM is a PGAS library
  - Interconnects (e.g. Infiniband, GNI, OPA); shared memory (e.g. knem, xpmem)
  - Point-to-point RDMA and Contexts
  - Teams and Collectives
  - Atomics, locks
  - Dynamic memory management
- Open specification
- Community driven
- Various implementations
  - SBU / OSSS, Open-MPI, SOS, MVAPICH2-X
  - OSHMPI
  - Vendor: Cray/IBM/…

http://www.openshmem.org/

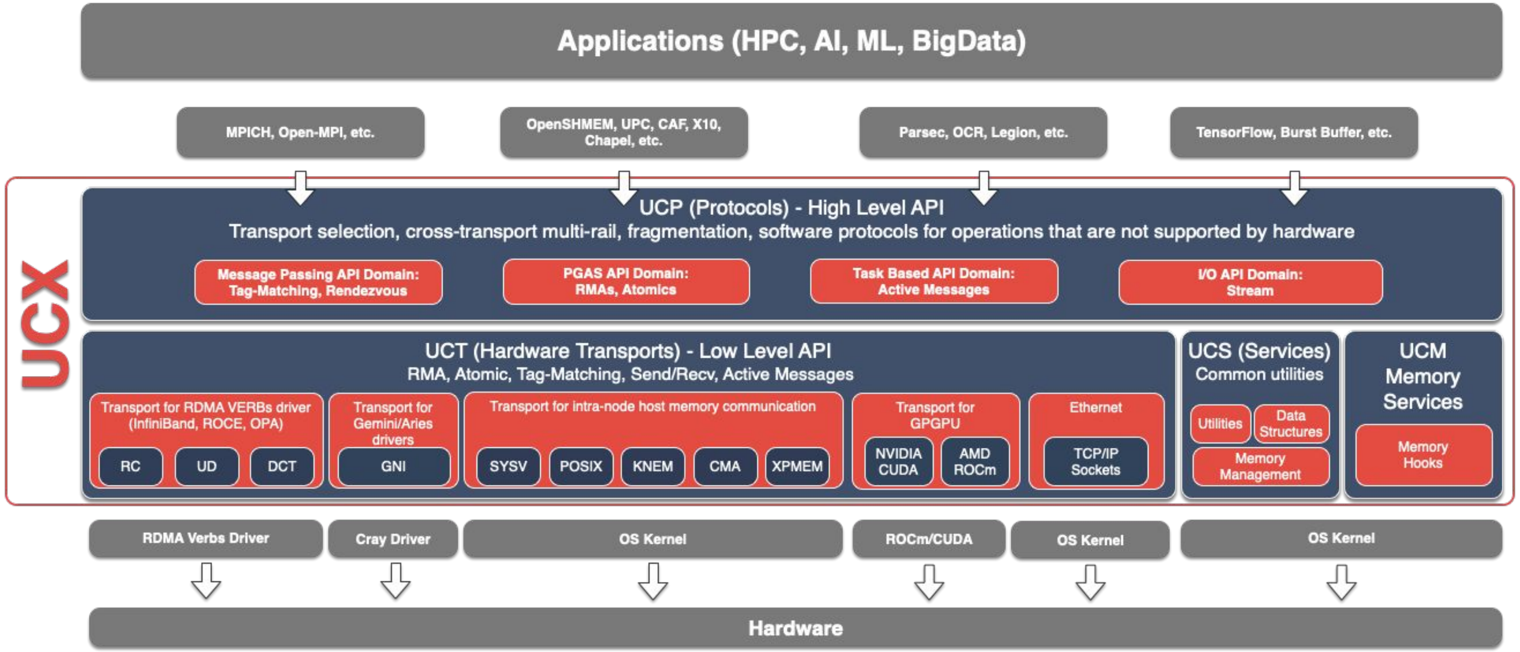
- Project
  - Funding: DOD / LANL
  - Interfacing with OpenSHMEM using newer languages:
    - Rust
    - Go!
  - Goals:
    - Memory Safety
    - Security
    - Speed
    - Usability

UCF2021: Rust & OpenSHMEM

- Reference Implementation
  - Communications: UCX
  - Wireup: PMIx
  - Collectives: SHCOLL (Duke/Rice collab)
    - Plan to move to UCC

iACS INSTITUTE FOR ADVANCED COMPUTATIONAL SCIENCE

UCF2021: Rust & OpenSHMEM


Applications (HPC, AI, ML, BigData)
MPICH, Open-MPI, etc.
OpenSHMEM, UPC, CAF, X10, Chapel, etc.
Parsec, OCR, Legion, etc.
TensorFlow, Burst Buffer, etc.
UCX
UCP (Protocols) - High Level API
Transport selection, cross-transport multi-rail, fragmentation, software protocols for operations that are not supported by hardware
Message Passing API Domain: Tag-Matching, Rendezvous
PGAS API Domain: RMAs, Atomics
Task Based API Domain: Active Messages
I/O API Domain: Stream
UCT (Hardware Transports) - Low Level API
RMA, Atomic, Tag-Matching, Send/Recv, Active Messages
Transport for RDMA VERBs driver (InfiniBand, ROCE, OPA)
RC
UD
DCT
Transport for Gemini/Aries drivers
GNI
Transport for intra-node host memory communication
SYSV
POSIX
KNEM
CMA
XPMEM
Transport for GPGPU
NVIDIA CUDA
AMD ROCm
Ethernet
TCP/IP Sockets
UCS (Services)
Common utilities
Utilities
Data Structures
Memory Management
UCM Memory Services
Memory Hooks
RDMA Verbs Driver
Cray Driver
OS Kernel
ROCm/CUDA
OS Kernel
OS Kernel
Hardware

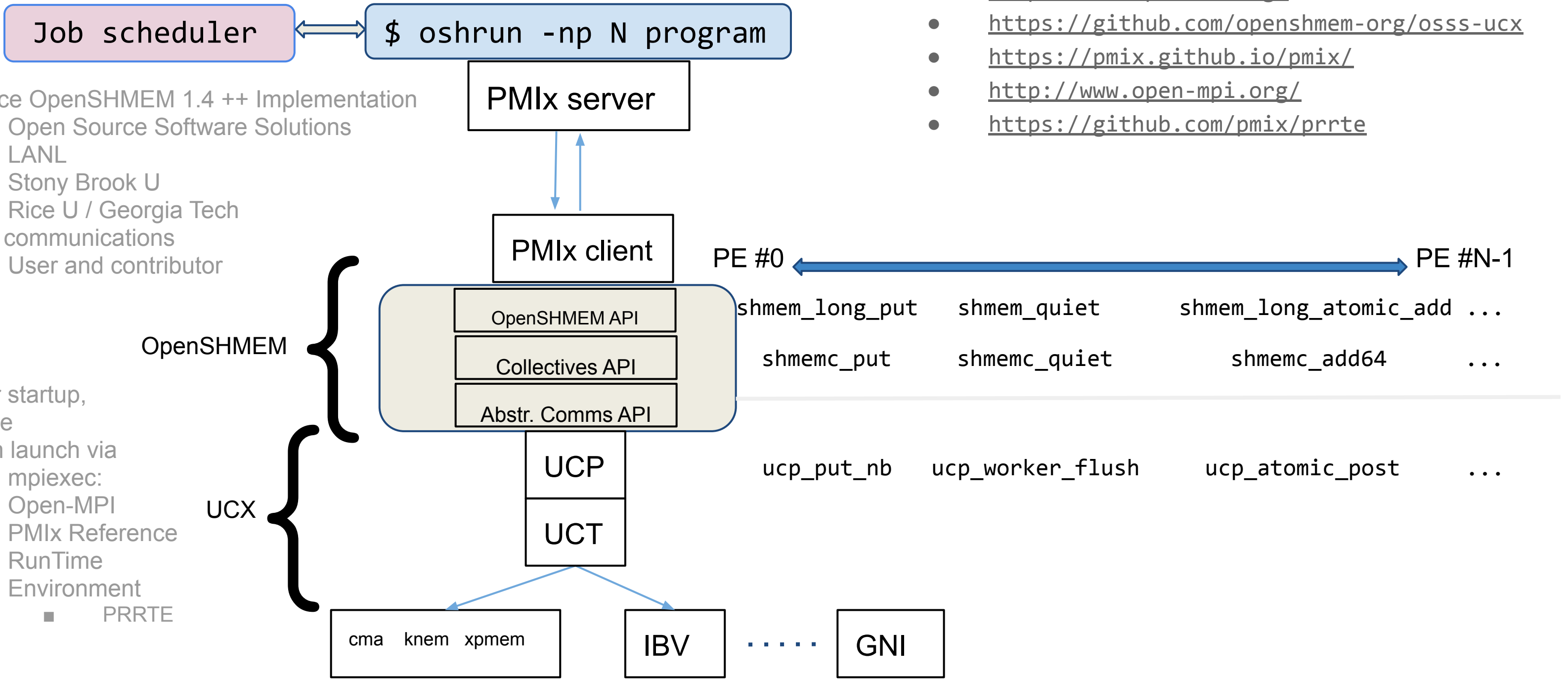
UCF2021: Rust & OpenSHMEM

Tony Curtis (SBU), Howard Pritchard (LANL)

http://www.openshmem.org/

- http://www.openucx.org/
- https://github.com/openshmem-org/osss-ucx
- https://pmix.github.io/pmix/
- http://www.open-mpi.org/
- https://github.com/pmix/prrte

- Reference OpenSHMEM 1.4 ++ Implementation
  - Open Source Software Solutions
  - LANL
  - Stony Brook U
  - Rice U / Georgia Tech
- UCX for communications
  - User and contributor
- PMIx for startup, resilience
- Program launch via
  - mpiexec: Open-MPI
  - PMIx Reference RunTime Environment
    - PRRTE



UCF2021: Rust & OpenSHMEM

Rust meets OpenSHMEM

Proof of Concept interface with OSHMEM

UCF2021: Rust & OpenSHMEM

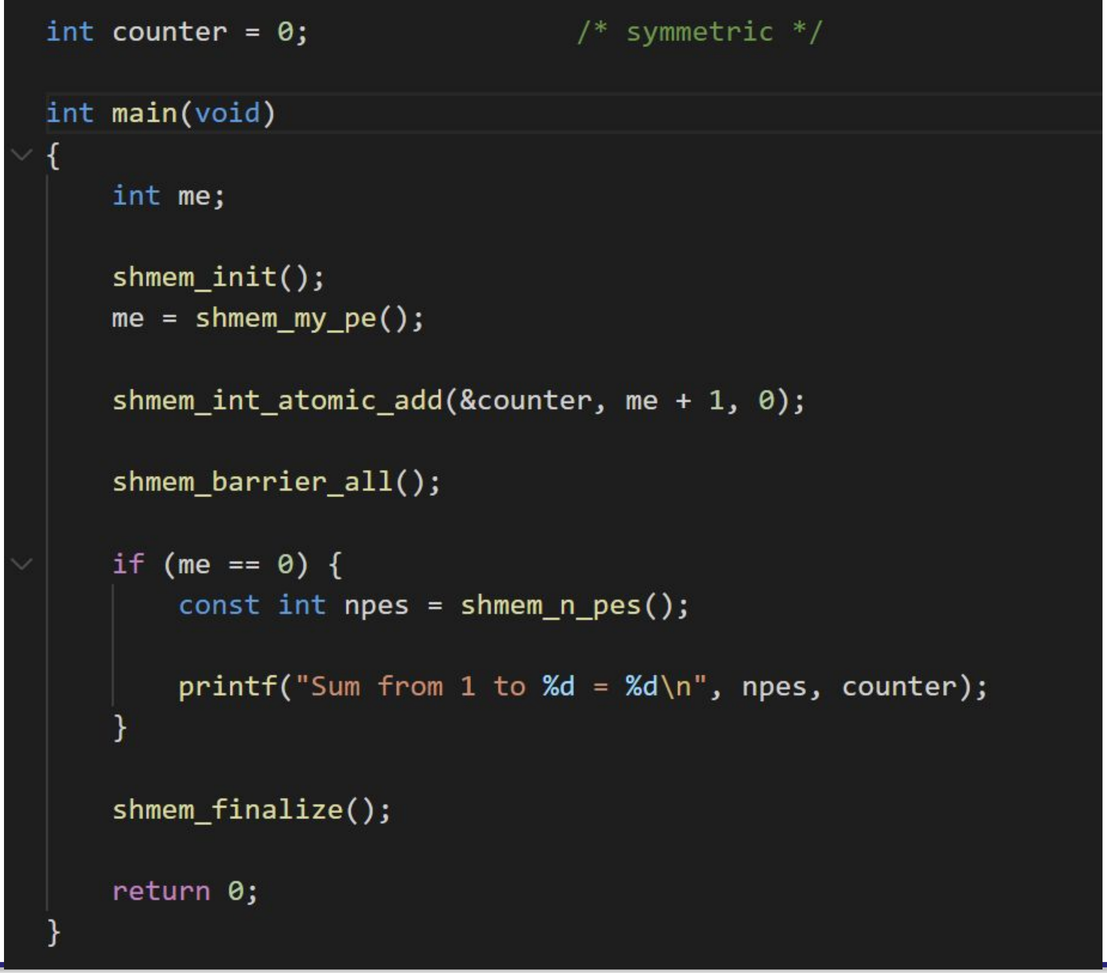
- Rust is a new language to sit where C/C++ used to
  - Looks a bit like Java/C++
  - Efforts to insert into Linux kernel & userland
  - https://doc.rust-lang.org/book/
  - Because it has useful safety guarantees (motivation)

Rust Prioritizes Safety

- Ownership rules and borrow checker prevent undefined behavior, e.g.
  - Dereferencing null or dangling pointers
  - Reading uninitialized memory
  - Data races
  - Use-After-Free

UCF2021: Rust & OpenSHMEM

C code with globals and raw pointers



```c
int counter = 0;                    /* symmetric */

int main(void)
{
    int me;

    shmem_init();
    me = shmem_my_pe();

    shmem_int_atomic_add(&counter, me + 1, 0);

    shmem_barrier_all();

    if (me == 0) {
        const int npes = shmem_n_pes();

        printf("Sum from 1 to %d = %d\n", npes, counter);
    }

    shmem_finalize();

    return 0;
}
```

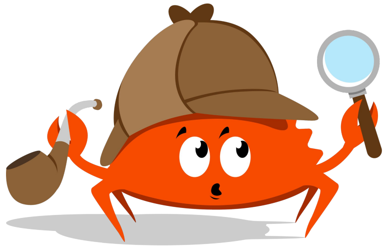
UCF2021: Rust & OpenSHMEM

```rust
use std::mem::drop; // equivalent to free()

fn main() {
  let x = "Hello".to_string();
  drop(x);
  println!("{}", x);
}
```

```
error[E0382]: use of moved value: `x`
 --> test.rs:6:18
  |
5 |   drop(x);
  |        - value moved here
6 |   println!("{}", x);
  |                  ^ value used here after move
  |
  = note: move occurs because `x` has type `std::string::String`, which does not implement the `Copy` trait
```

INSTITUTE FOR ADVANCED COMPUTATIONAL SCIENCE

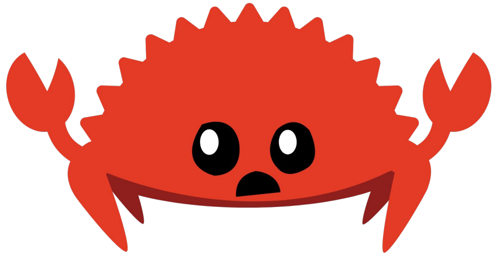
Unsafe Rust Superpowers



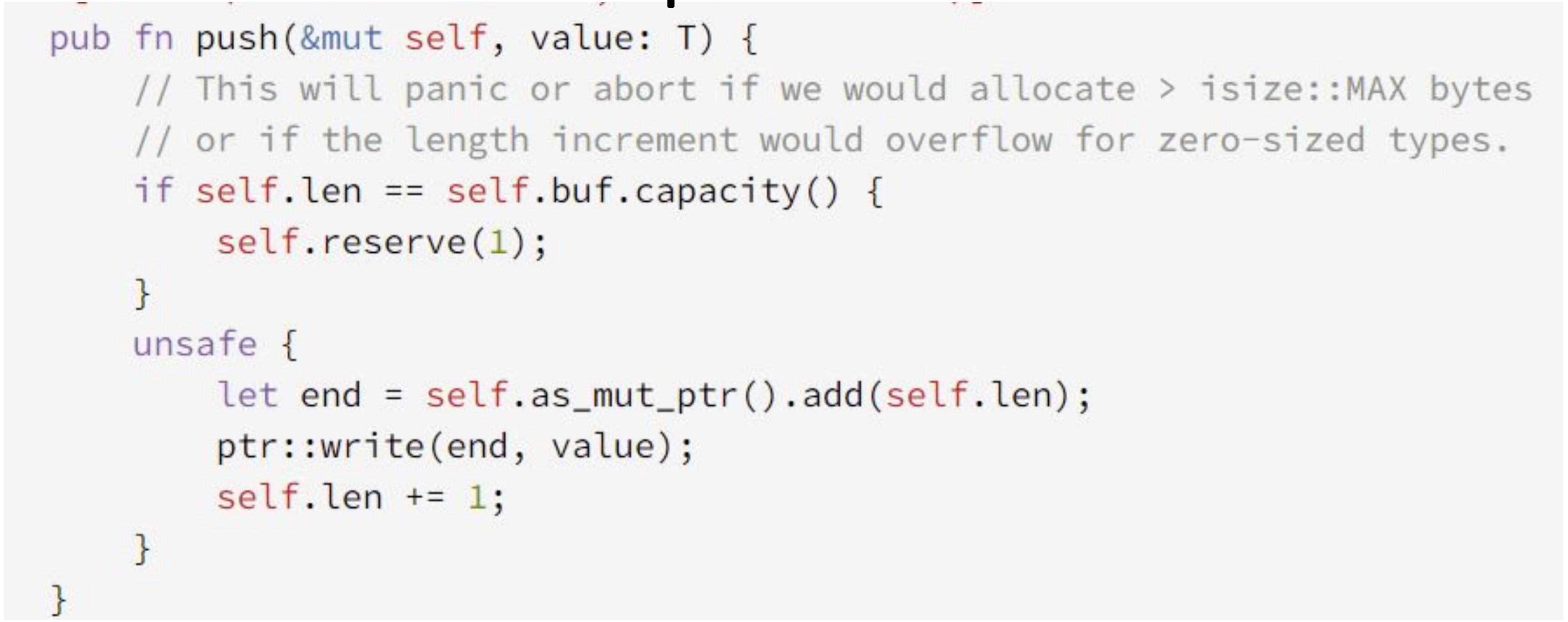
- Dereferencing raw pointers
- Calling unsafe functions (e.g. C FFI)
- Accessing/Modifying mutable static variables
- Implement unsafe traits
- Accessing union fields

UCF2021: Rust & OpenSHMEM

Rust Programming might appear safe at first glance.. but don't look under the hood. You'll get a nasty surprise!



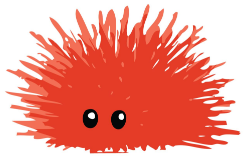
std::vec::Vec Implementation



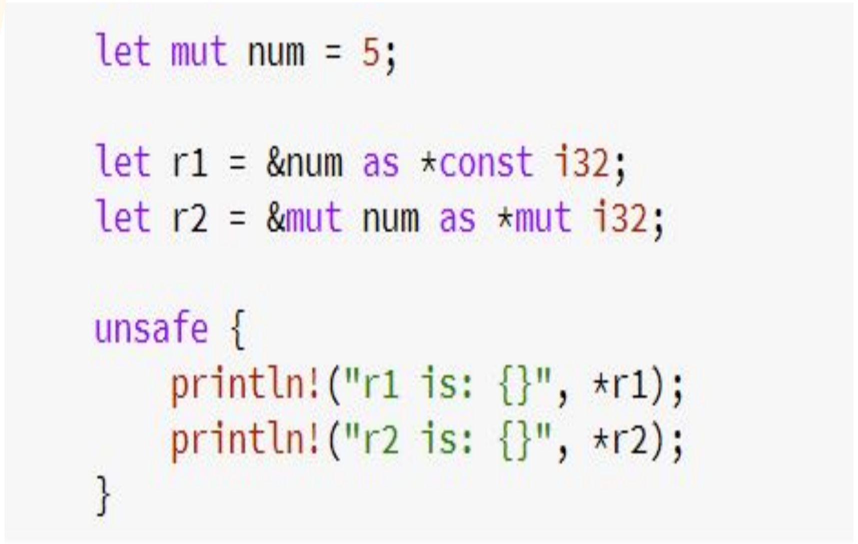
```
pub fn push(&mut self, value: T) {
    // This will panic or abort if we would allocate > isize::MAX bytes
    // or if the length increment would overflow for zero-sized types.
    if self.len == self.buf.capacity() {
        self.reserve(1);
    }
    unsafe {
        let end = self.as_mut_ptr().add(self.len);
        ptr::write(end, value);
        self.len += 1;
    }
}
```

iACS INSTITUTE FOR ADVANCED COMPUTATIONAL SCIENCE

Raw Pointers

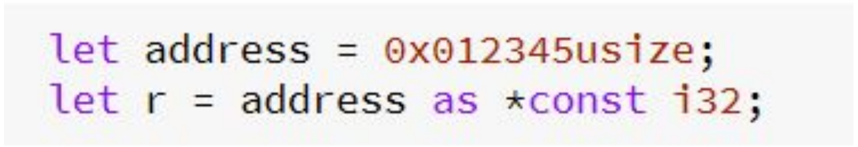


Valid



```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

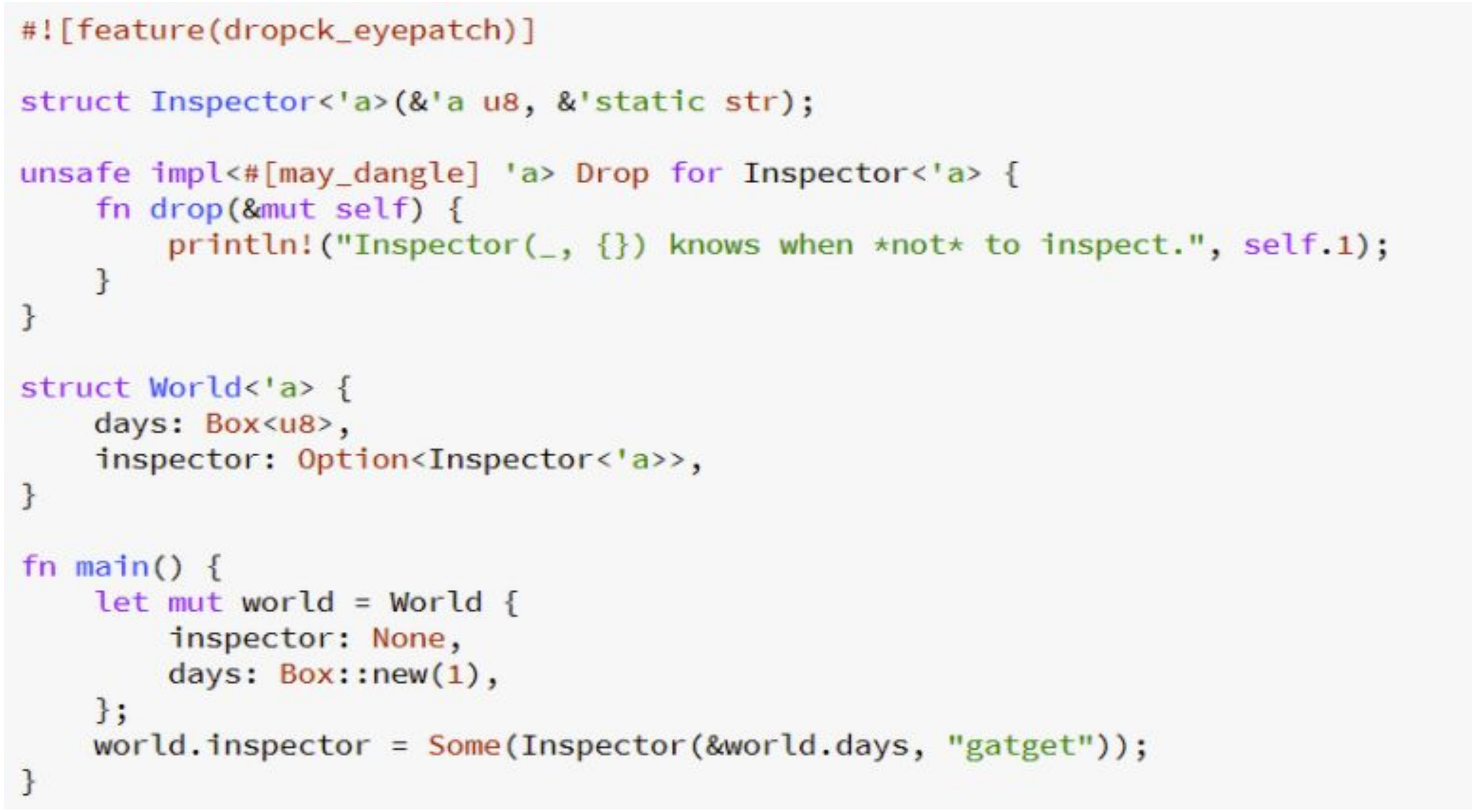
Invalid



```
let address = 0x012345usize;
let r = address as *const i32;
```

Dereferencing arbitrary memory location is undefined behavior

Unsafe Drop Example

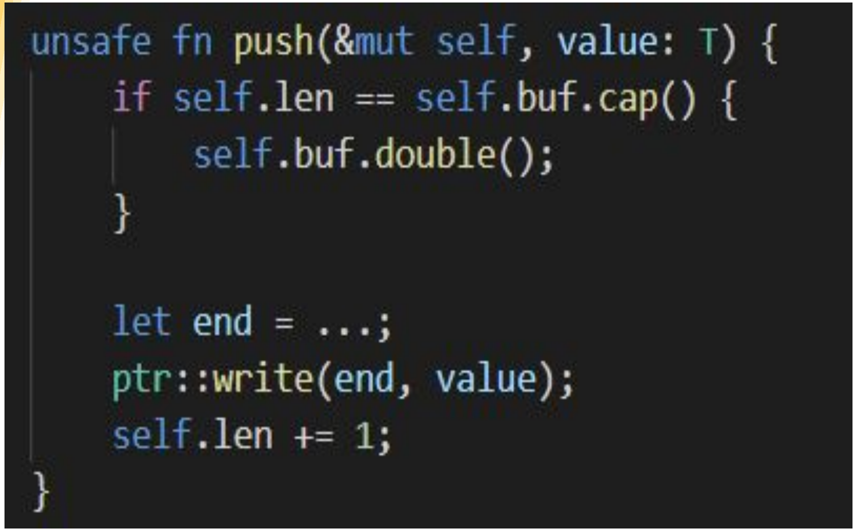


```rust
#![feature(dropck_eyepatch)]

struct Inspector<'a>(&'a u8, &'static str);

unsafe impl<#[may_dangle] 'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("Inspector(_, {}) knows when *not* to inspect.", self.1);
    }
}

struct World<'a> {
    days: Box<u8>,
    inspector: Option<Inspector<'a>>,
}

fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days, "gatget"));
}
```

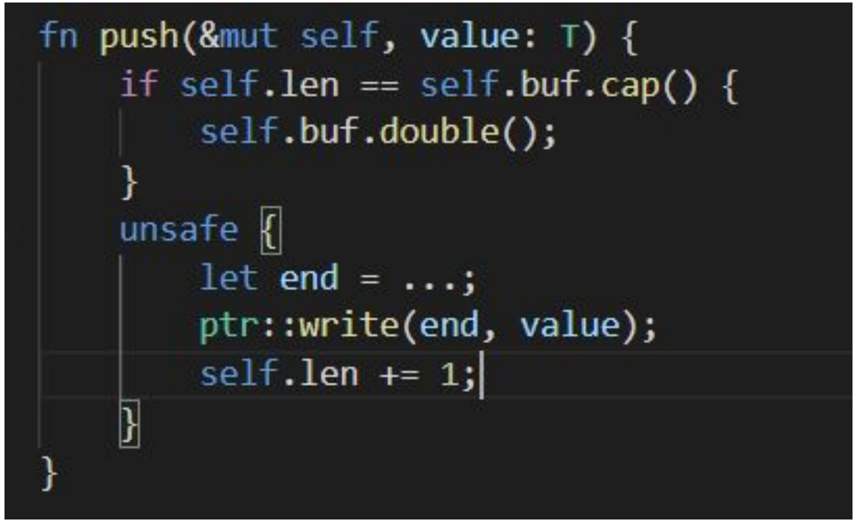
UCF2021: Rust & OpenSHMEM
Dynamic Array Initialization

```rust
use std::mem::{self, MaybeUninit};

// Size of the array is hard-coded but easy to change (meaning, changing just
// the constant is sufficient). This means we can't use [a, b, c] syntax to
// initialize the array, though, as we would have to keep that in sync
// with `SIZE`!
const SIZE: usize = 10;

let x = {
    // Create an uninitialized array of `MaybeUninit`. The `assume_init` is
    // safe because the type we are claiming to have initialized here is a
    // bunch of `MaybeUninit`s, which do not require initialization.
    let mut x: [MaybeUninit<Box<u32>>; SIZE] = unsafe {
        MaybeUninit::uninit().assume_init()
    };

    // Dropping a `MaybeUninit` does nothing. Thus using raw pointer
    // assignment instead of `ptr::write` does not cause the old
    // uninitialized value to be dropped.
    // Exception safety is not a concern because Box can't panic
    for i in 0..SIZE {
        x[i] = MaybeUninit::new(Box::new(i as u32));
    }

    // Everything is initialized. Transmute the array to the
    // initialized type.
    unsafe { mem::transmute::<_, [Box<u32>; SIZE]>(x) }
};
```

Unsafe Functions vs. Unsafe Blocks

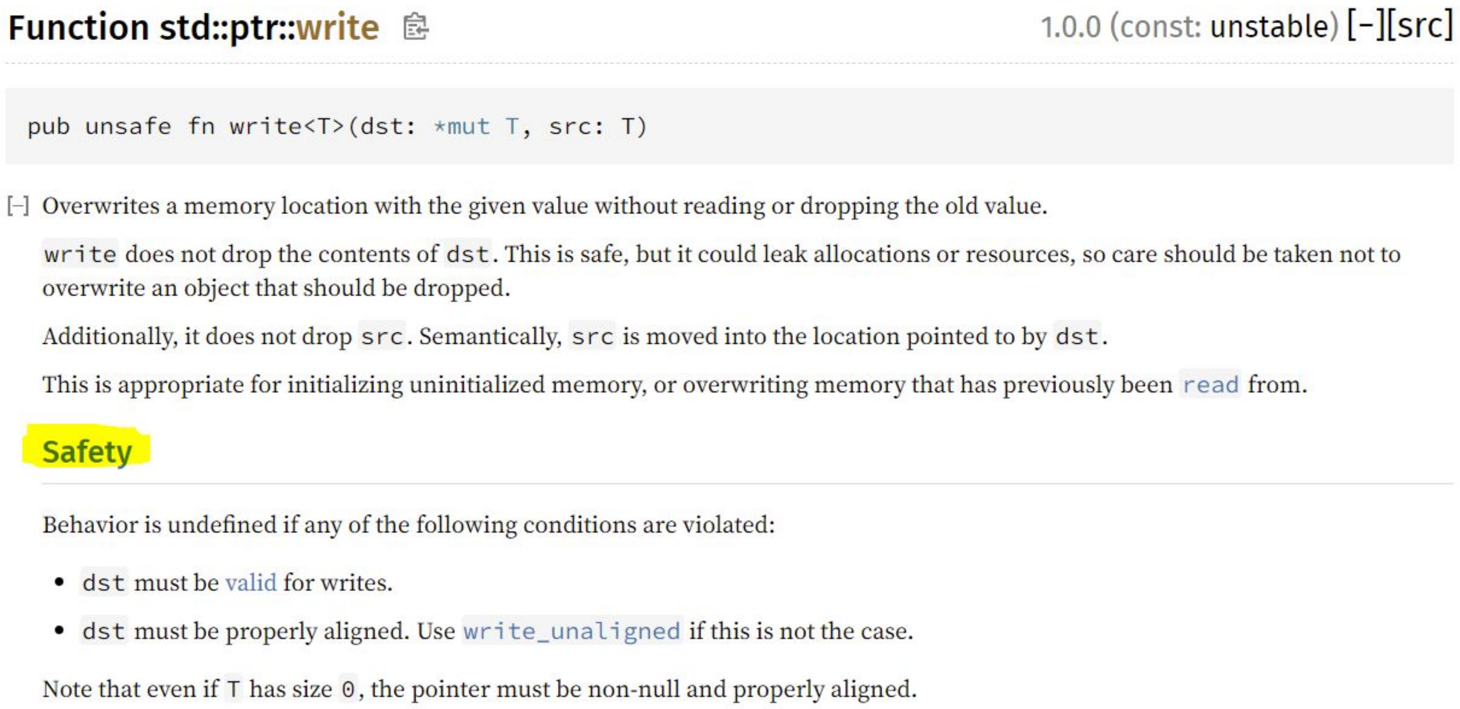


```rust
unsafe fn push(&mut self, value: T) {
    if self.len == self.buf.cap() {
        self.buf.double();
    }

    let end = ...;
    ptr::write(end, value);
    self.len += 1;
}
```



```rust
fn push(&mut self, value: T) {
    if self.len == self.buf.cap() {
        self.buf.double();
    }
    unsafe {
        let end = ...;
        ptr::write(end, value);
        self.len += 1;
    }
}
```

UCF2021: Rust & OpenSHMEM
Unsafe Function Contracts



Function std::ptr::write

1.0.0 (const: unstable) [−][src]

```
pub unsafe fn write<T>(dst: *mut T, src: T)
```

[−] Overwrites a memory location with the given value without reading or dropping the old value.

`write` does not drop the contents of `dst`. This is safe, but it could leak allocations or resources, so care should be taken not to overwrite an object that should be dropped.

Additionally, it does not drop `src`. Semantically, `src` is moved into the location pointed to by `dst`.

This is appropriate for initializing uninitialized memory, or overwriting memory that has previously been `read` from.

Safety

Behavior is undefined if any of the following conditions are violated:

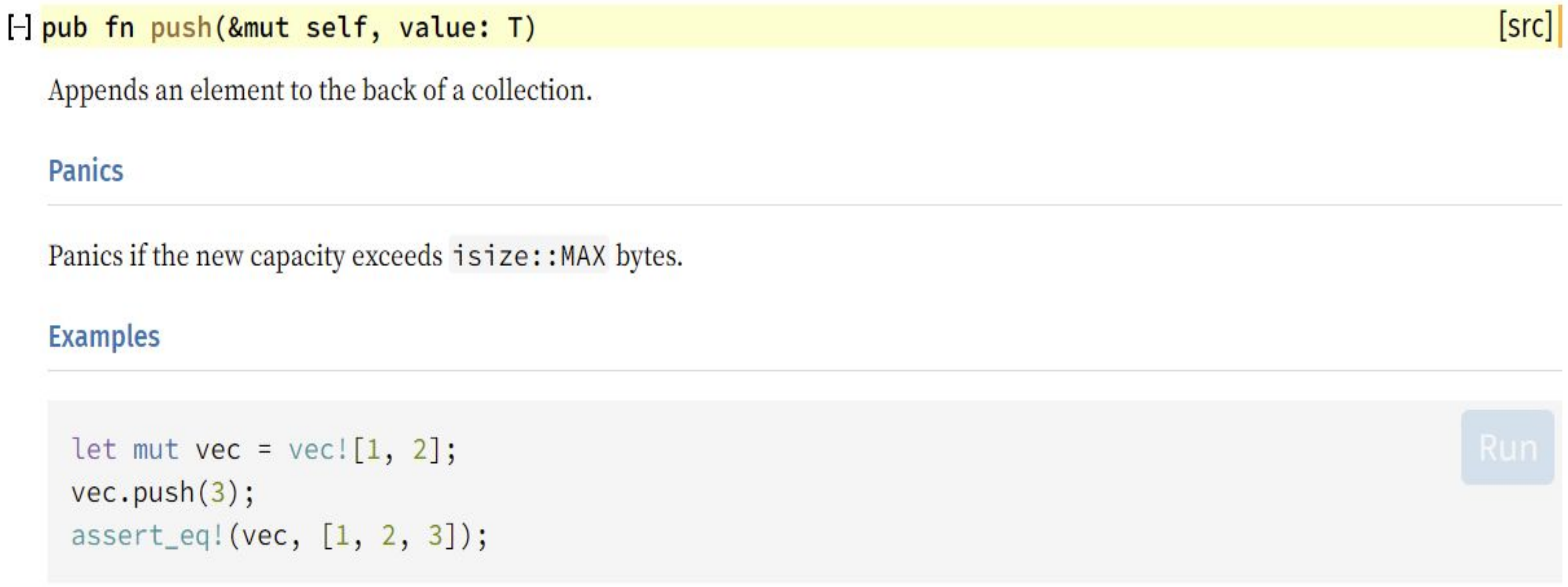
- `dst` must be valid for writes.
- `dst` must be properly aligned. Use `write_unaligned` if this is not the case.

Note that even if `T` has size `0`, the pointer must be non-null and properly aligned.

iACS INSTITUTE FOR ADVANCED COMPUTATIONAL SCIENCE

UCF2021: Rust & OpenSHMEM

Unsafe Function Contracts



[-] pub fn push(&mut self, value: T) [src]

Appends an element to the back of a collection.

Panics

Panics if the new capacity exceeds `isize::MAX` bytes.

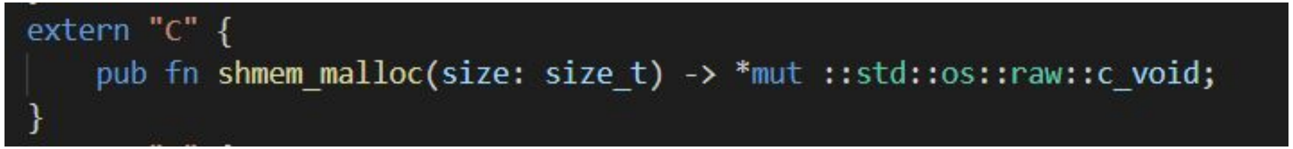
Examples

```
let mut vec = vec![1, 2];
vec.push(3);
assert_eq!(vec, [1, 2, 3]);
```

UCF2021: Rust & OpenSHMEM

Generating FFI

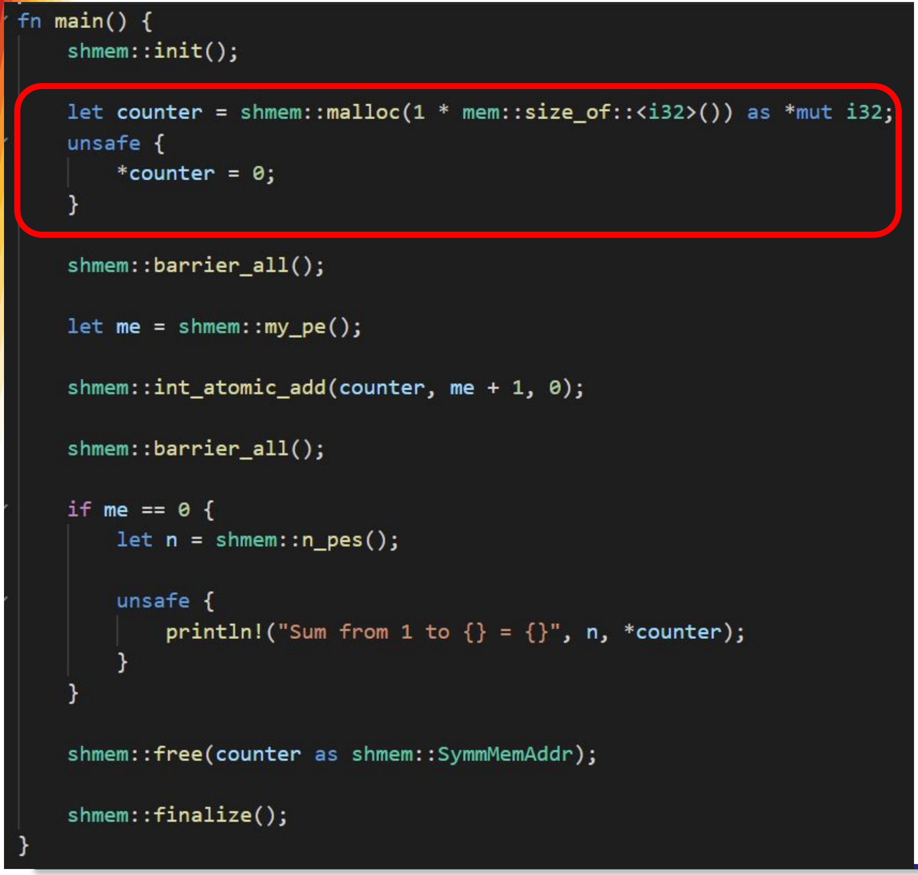
- rust-bindgen generates interface to call C/C++ library functions
  - libclang parses and type checks C/C++ header files



```
extern "C" {
    pub fn shmem_malloc(size: size_t) -> *mut ::std::os::raw::c_void;
}
```

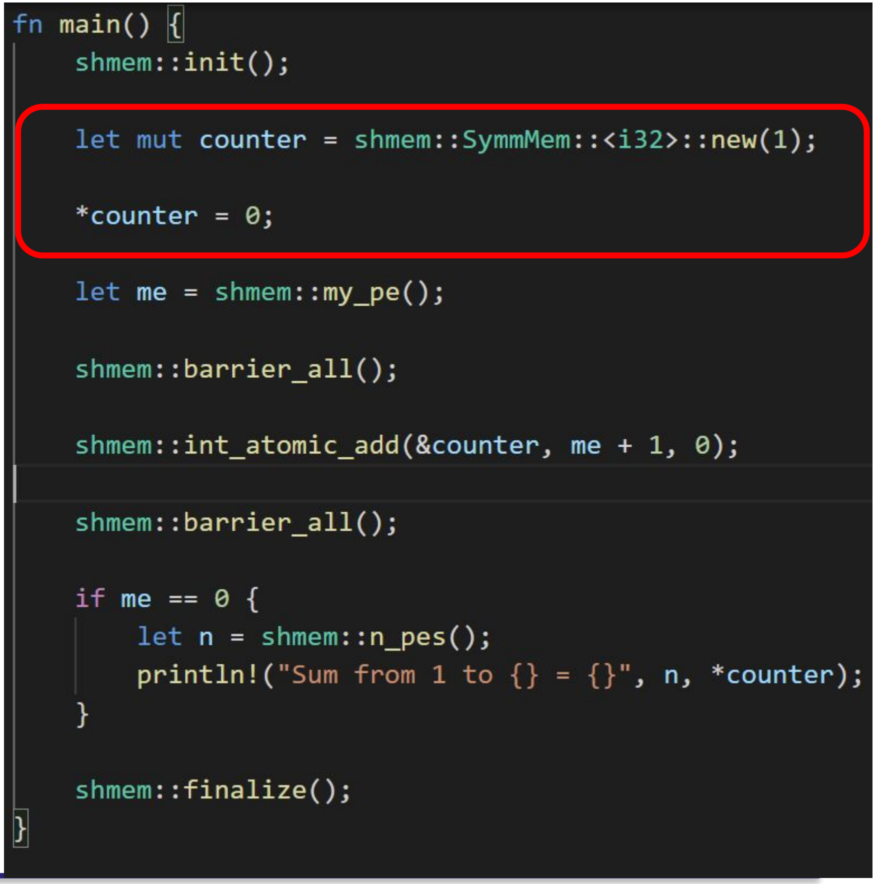
- All FFI functions are unsafe

Before:

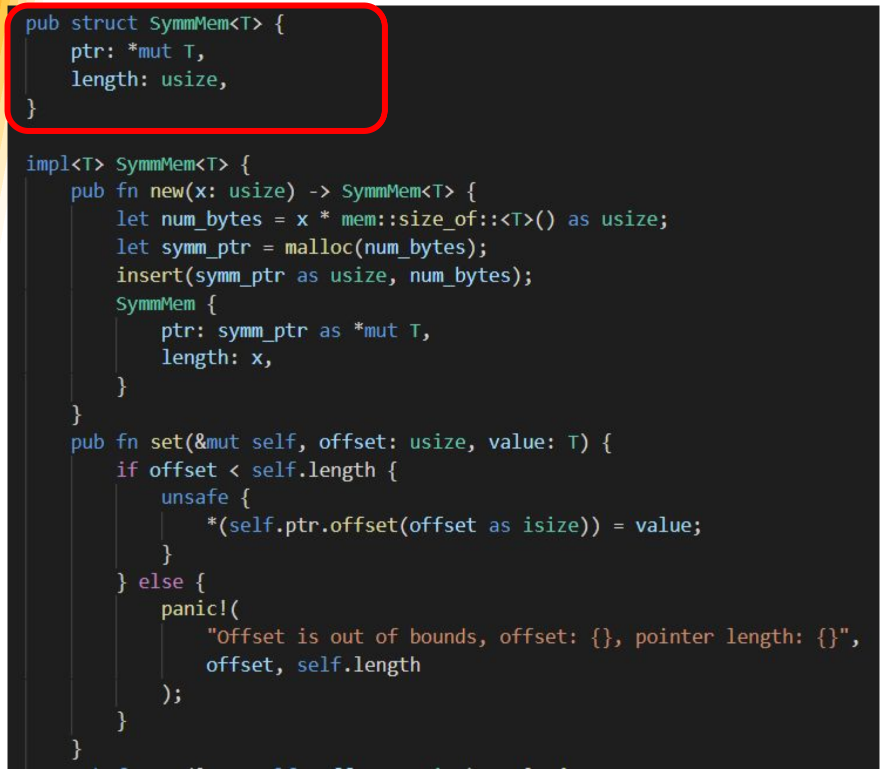


```rust
fn main() {
    shmem::init();

    let counter = shmem::malloc(1 * mem::size_of::<i32>()) as *mut i32;
    unsafe {
        *counter = 0;
    }

    shmem::barrier_all();

    let me = shmem::my_pe();

    shmem::int_atomic_add(counter, me + 1, 0);

    shmem::barrier_all();

    if me == 0 {
        let n = shmem::n_pes();

        unsafe {
            println!("Sum from 1 to {} = {}", n, *counter);
        }
    }

    shmem::free(counter as shmem::SymmMemAddr);

    shmem::finalize();
}
```

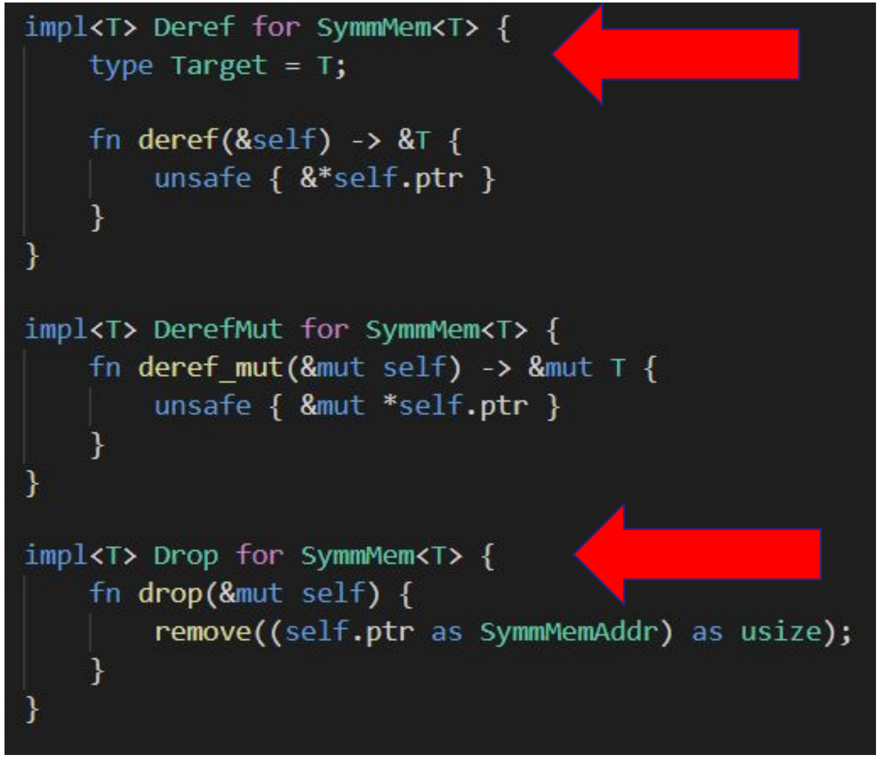
After:



```rust
fn main() {
    shmem::init();

    let mut counter = shmem::SymmMem::<i32>::new(1);

    *counter = 0;

    let me = shmem::my_pe();

    shmem::barrier_all();

    shmem::int_atomic_add(&counter, me + 1, 0);

    shmem::barrier_all();

    if me == 0 {
        let n = shmem::n_pes();
        println!("Sum from 1 to {} = {}", n, *counter);
    }

    shmem::finalize();
}
```

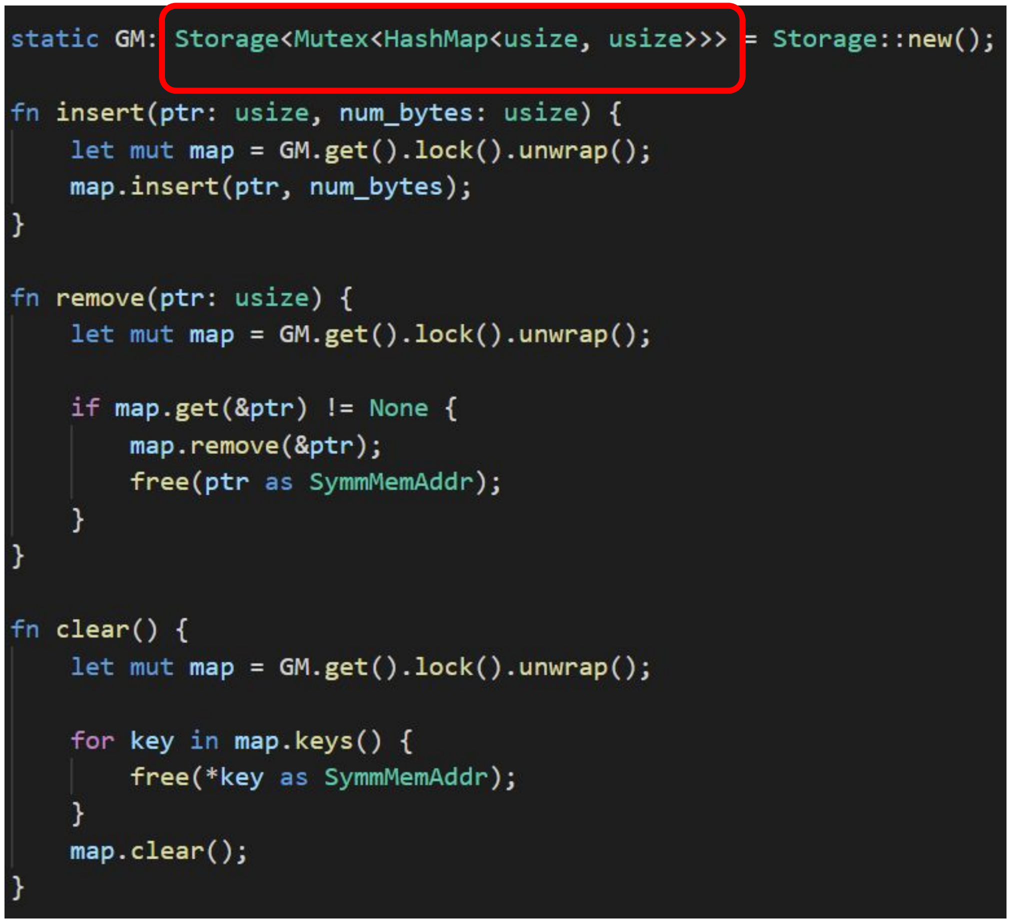


```rust
pub struct SymmMem<T> {
    ptr: *mut T,
    length: usize,
}

impl<T> SymmMem<T> {
    pub fn new(x: usize) -> SymmMem<T> {
        let num_bytes = x * mem::size_of::<T>() as usize;
        let symm_ptr = malloc(num_bytes);
        insert(symm_ptr as usize, num_bytes);
        SymmMem {
            ptr: symm_ptr as *mut T,
            length: x,
        }
    }
    pub fn set(&mut self, offset: usize, value: T) {
        if offset < self.length {
            unsafe {
                *(self.ptr.offset(offset as isize)) = value;
            }
        } else {
            panic!(
                "Offset is out of bounds, offset: {}, pointer length: {}",
                offset, self.length
            );
        }
    }
```



```rust
impl<T> Deref for SymmMem<T> {
    type Target = T;

    fn deref(&self) -> &T {
        unsafe { &*self.ptr }
    }
}

impl<T> DerefMut for SymmMem<T> {
    fn deref_mut(&mut self) -> &mut T {
        unsafe { &mut *self.ptr }
    }
}

impl<T> Drop for SymmMem<T> {
    fn drop(&mut self) {
        remove((self.ptr as SymmMemAddr) as usize);
    }
}
```

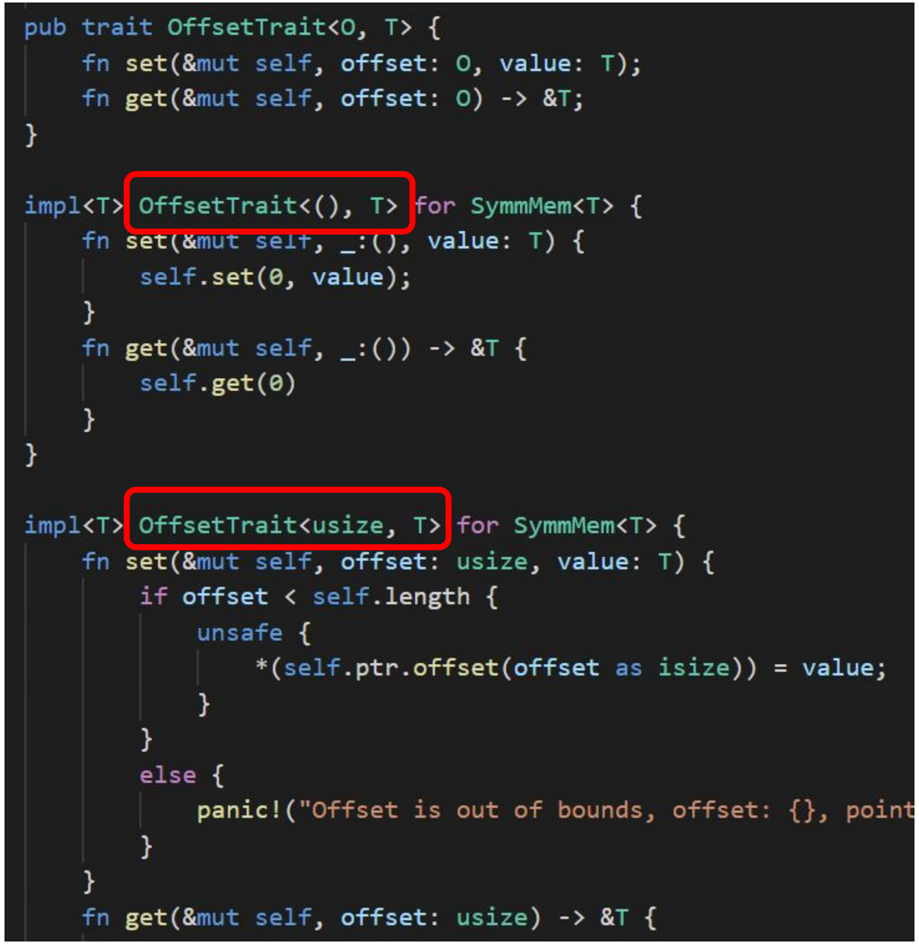
UCF2021: Rust & OpenSHMEM



```
static GM: Storage<Mutex<HashMap<usize, usize>>> = Storage::new();

fn insert(ptr: usize, num_bytes: usize) {
    let mut map = GM.get().lock().unwrap();
    map.insert(ptr, num_bytes);
}

fn remove(ptr: usize) {
    let mut map = GM.get().lock().unwrap();

    if map.get(&ptr) != None {
        map.remove(&ptr);
        free(ptr as SymmMemAddr);
    }
}

fn clear() {
    let mut map = GM.get().lock().unwrap();

    for key in map.keys() {
        free(*key as SymmMemAddr);
    }
    map.clear();
}
```

INSTITUTE FOR ADVANCED COMPUTATIONAL SCIENCE

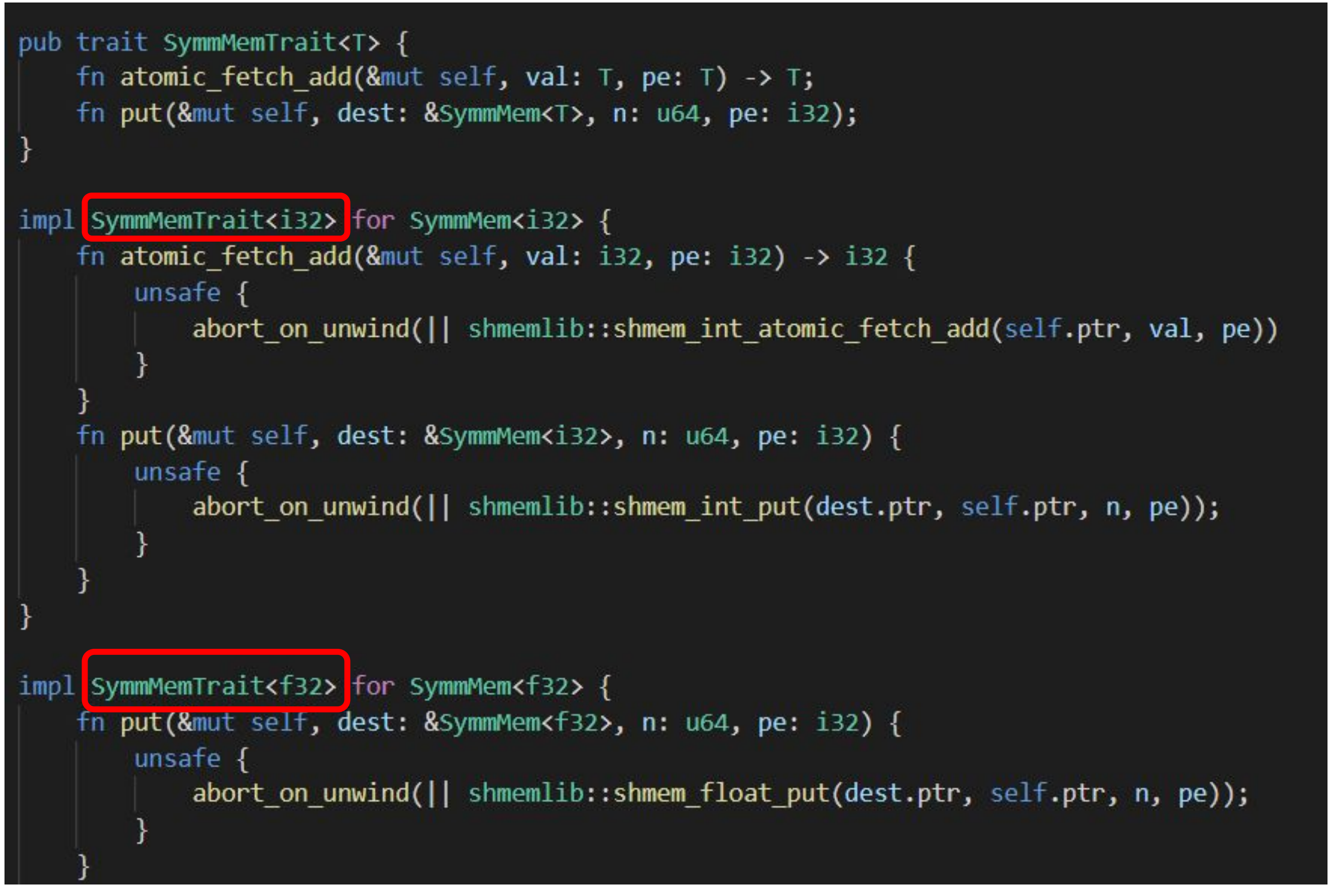
UCF2021: Rust & OpenSHMEM



```
pub trait OffsetTrait<O, T> {
    fn set(&mut self, offset: O, value: T);
    fn get(&mut self, offset: O) -> &T;
}

impl<T> OffsetTrait<(), T> for SymmMem<T> {
    fn set(&mut self, _:(), value: T) {
        self.set(0, value);
    }
    fn get(&mut self, _:()) -> &T {
        self.get(0)
    }
}

impl<T> OffsetTrait<usize, T> for SymmMem<T> {
    fn set(&mut self, offset: usize, value: T) {
        if offset < self.length {
            unsafe {
                *(self.ptr.offset(offset as isize)) = value;
            }
        }
        else {
            panic!("Offset is out of bounds, offset: {}, point
        }
    }
    fn get(&mut self, offset: usize) -> &T {
```

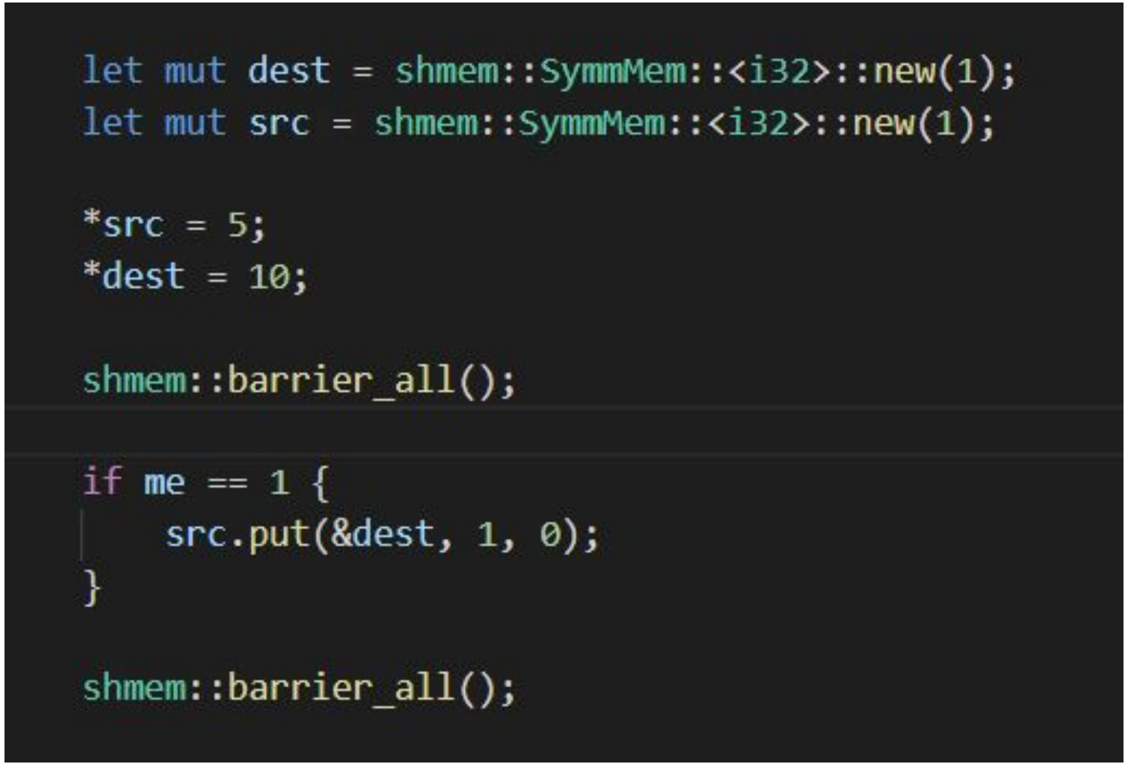
INSTITUTE FOR ADVANCED COMPUTATIONAL SCIENCE

UCF2021: Rust & OpenSHMEM



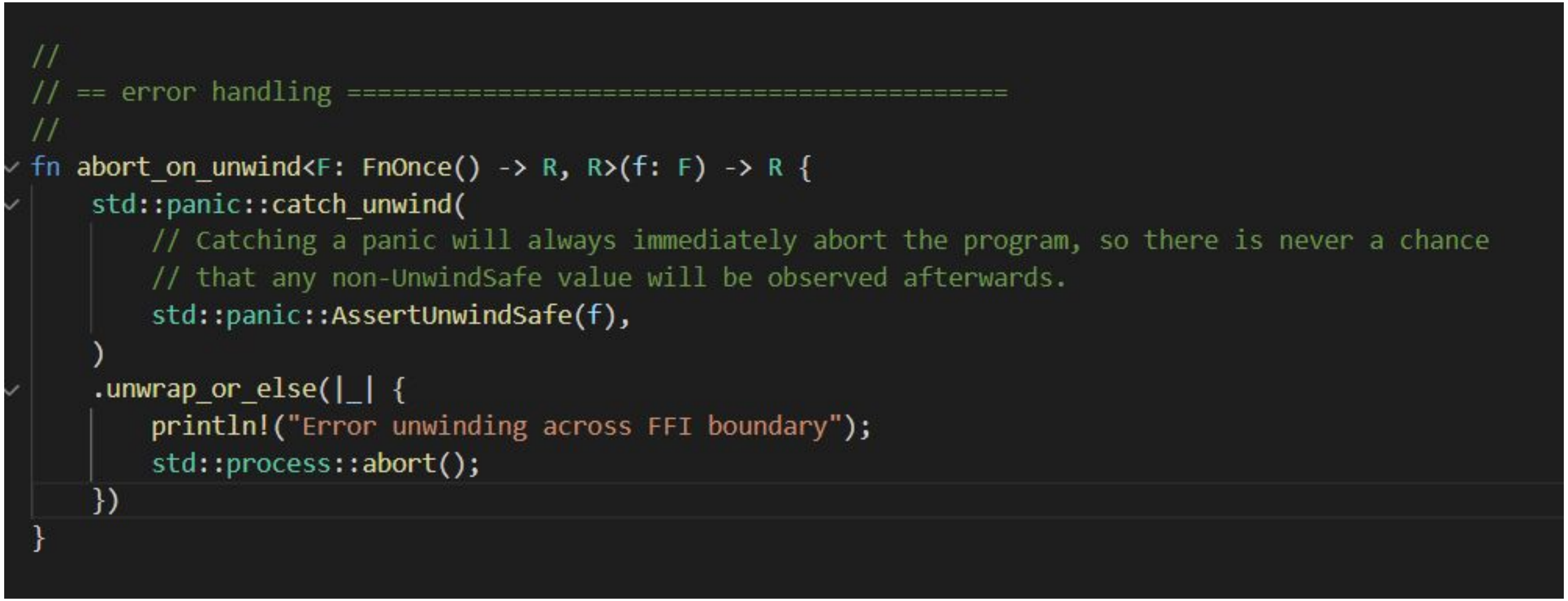
```
pub trait SymmMemTrait<T> {
    fn atomic_fetch_add(&mut self, val: T, pe: T) -> T;
    fn put(&mut self, dest: &SymmMem<T>, n: u64, pe: i32);
}

impl SymmMemTrait<i32> for SymmMem<i32> {
    fn atomic_fetch_add(&mut self, val: i32, pe: i32) -> i32 {
        unsafe {
            abort_on_unwind(|| shmemlib::shmem_int_atomic_fetch_add(self.ptr, val, pe))
        }
    }
    fn put(&mut self, dest: &SymmMem<i32>, n: u64, pe: i32) {
        unsafe {
            abort_on_unwind(|| shmemlib::shmem_int_put(dest.ptr, self.ptr, n, pe));
        }
    }
}

impl SymmMemTrait<f32> for SymmMem<f32> {
    fn put(&mut self, dest: &SymmMem<f32>, n: u64, pe: i32) {
        unsafe {
            abort_on_unwind(|| shmemlib::shmem_float_put(dest.ptr, self.ptr, n, pe));
        }
    }
```

UCF2021: Rust & OpenSHMEM



```
let mut dest = shmem::SymmMem::<i32>::new(1);
let mut src = shmem::SymmMem::<i32>::new(1);

*src = 5;
*dest = 10;

shmem::barrier_all();

if me == 1 {
    src.put(&dest, 1, 0);
}

shmem::barrier_all();
```

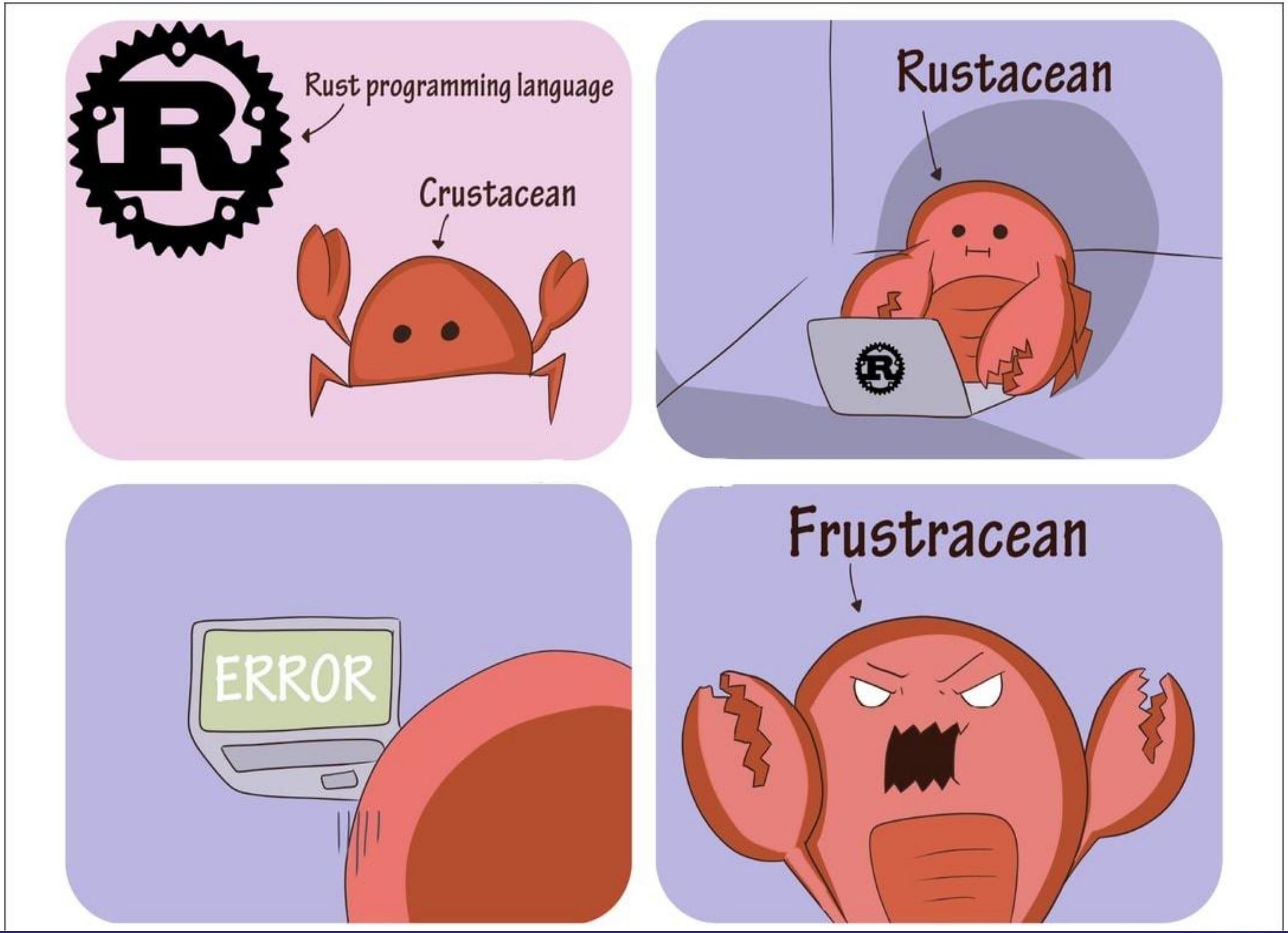
UCF2021: Rust & OpenSHMEM



```
//
// == error handling ==========================================
//
fn abort_on_unwind<F: FnOnce() -> R, R>(f: F) -> R {
    std::panic::catch_unwind(
        // Catching a panic will always immediately abort the program, so there is never a chance
        // that any non-UnwindSafe value will be observed afterwards.
        std::panic::AssertUnwindSafe(f),
    )
    .unwrap_or_else(|_| {
        println!("Error unwinding across FFI boundary");
        std::process::abort();
    })
}
```

UCF2021: Rust & OpenSHMEM
Future Work

- Generic Functions
  - Infer SymmMem struct parameter type using reflection
- Assessing Rust’s FFI Overhead
- Direct Rust/UCP interface


Rust programming language
Crustacean
Rustacean
ERROR
Frustracean

UCF2021: Rust & OpenSHMEM

- Thanks to
  - DoD/LANL/OSSS/SBU and all project partners
  - NSF for SBU’s ookami cluster
    - https://www.stonybrook.edu/ookami/
  - And of course, Rebecca, for wading bravely into this project