0x15 Programming Language Exercise Session

hexhive

EPFL

Recap

Type Safety

*A type system is a logical system that assigns a **type** to every **term** (variable, expression, function)*

- Static vs. Dynamic
- Declared vs. Inferred
- Strict vs. Relaxed

What are the Pros and Cons of these features?

Memory Safety

*Memory safety ensures that only **valid** objects are accessed **in bounds***

Spatial memory safety: objects are only accessed in bounds

Temporal memory safety: only valid objects are accessed

Automatic Storage Reclamation (Pros and Cons)

- Garbage collection
- Reference counting
- Smart pointers/borrow checking

Thread Safety

Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction.

Examples of concurrency bugs

- Race Condition
- Deadlock

Synchronization primitives and language constructs

- Locks/Semaphores/Atomic variables
- Java `synchronized` keyword
- Goroutines, etc.

Sandboxing and Compartmentalization

Sandboxing *means running programs (often untrusted code) in an isolated, restricted environment*

Implemented by restricting access to certain library/system calls

- `seccomp (SELinux)`
- `cgroup (Docker)`

Compartmentalization *breaks a complex system into small components and limit the access of entities to only what is necessary.*

Recall PoLP

Demo: C vs Rust

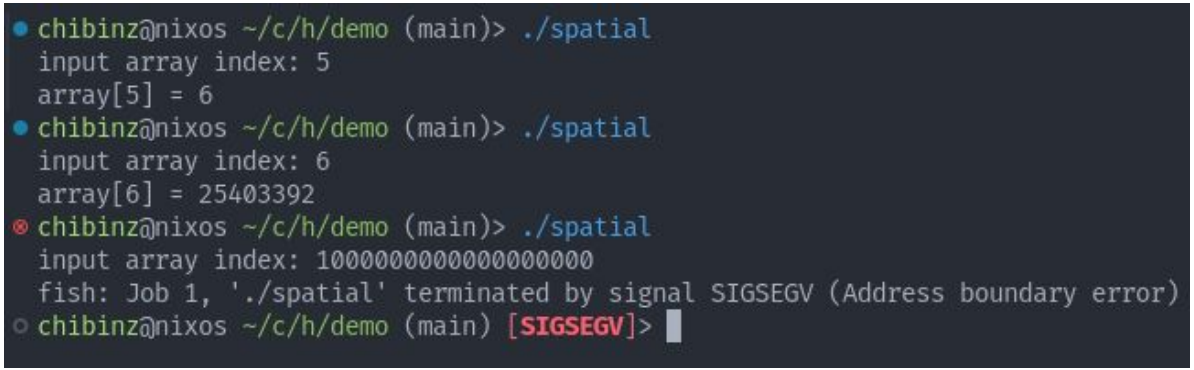
Spatial Memory Safety

```c
#include <stdio.h>

int main() {
    int index;
    int array[] = {1, 2, 3, 4, 5, 6};

    printf("input array index: ");
    scanf("%d", &index);
    printf("array[%d] = %d\n", index,
array[index]);

    return 0;
}
```

```rust
use scanf::scanf;

fn main() {
    let mut index = 0;
    let array = [1, 2, 3, 4, 5, 6];

    print!("input array index: ");
    scanf!("{}", index).unwrap();

    println!("array[{}] = {}", index,
array[index]);
}
```

`index` is an variable controlled by the user, what if it's >= 6…?

Spatial Memory Safety



C does not check bounds, may read garbage from stack, crash, or worse (undefined behavior)

Spatial Memory Safety

```
chibinz@nixos ~/c/h/demo (main)> ./spatial
input array index: 5
array[5] = 6
chibinz@nixos ~/c/h/demo (main)> ./spatial
input array index: 6
array[6] = 25403392
chibinz@nixos ~/c/h/demo (main)
input array index: 100000000000
fish: Job 1, './spatial' termin
chibinz@nixos ~/c/h/demo (main)
```

C does not
garbage fro
(und

```
chibinz@nixos ~/c/h/demo (main)> rust/target/debug/spatial
input array index: 5
array[5] = 6
chibinz@nixos ~/c/h/demo (main)> rust/target/debug/spatial
input array index: 6
thread 'main' panicked at src/spatial.rs:10:39:
index out of bounds: the len is 6 but the index is 6
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
chibinz@nixos ~/c/h/demo (main) [101]>
```

Rust explicitly check bounds, and terminates program when index is out of bound

Temporal Memory Safety

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr =  malloc(sizeof(int));
    *ptr = 1;
    printf("*ptr = %d\n", *ptr);
    free(ptr);
    printf("ptr freed\n");
    printf("try to access ptr again,
 *ptr = %d\n", *ptr);
    return 0;
}
```

```rust
fn main() {
    let boxed = Box::new(1);

    println!("boxed value: {}", boxed);
    drop(boxed);

    println!("dropping boxed ptr");
    println!("access boxed after drop: {}",
boxed);
}
```

Use after free is a frequent issue in C. In Rust, we can force a free by using `drop`.

Temporal Memory Safety

```
chibinz@nixos ~/c/h/demo (main) [101]> ./temporal
*ptr = 1
ptr freed
try to access ptr again, *ptr = 8728
```

Again, C assumes object is valid, and reads garbage from heap.

Temporal Memory Safety

```
chibinz@nixos ~/c/h/demo (main) [101]> ./temporal
*ptr = 1
ptr freed
try to access ptr again, *ptr = 8728
```

Again, C assumes object is valid, and reads garbage fro

```
error[E0382]: borrow of moved value: `boxed`
  --> src/temporal.rs:10:45
   |
2  |     let boxed = Box::new(1);
   |         ----- move occurs because `boxed` has type `Box<i32>`, which does not implement the `Copy` trait
...
6  |     drop(boxed);
   |          ----- value moved here
...
10 |     println!("access boxed after drop: {}", boxed);
   |                                             ^^^^^ value borrowed here after move
   |
```

Rust catches the issue at compile time, and complains we cannot use a freed (moved) value

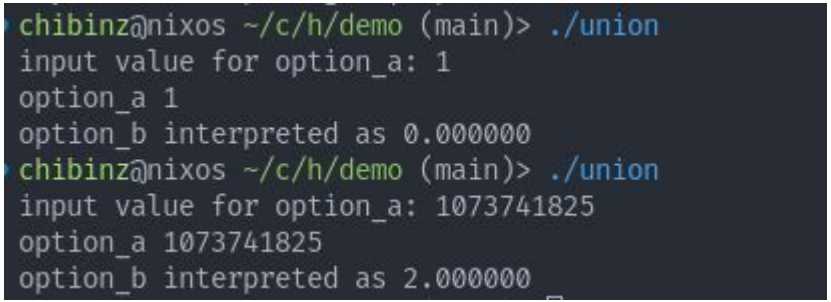
Type Safety

```c
union options {
  int option_a;
  float option_b;
};

int main() {
  union options o;
  printf("input value for option_a: ");
  scanf("%d", &o.option_a);
  printf("option_a %i\n", o.option_a);
  printf("option_b interpreted as %f\n",
o.option_b);
}
```

```rust
enum Options {
    A(i32),
    B(f32),
}

fn main() {
    let mut i = 0;
    print!("input value for Option::A: ");
    scanf!("{}", i).unwrap();

    let o = Options::A(i);
    println!("o.A {}", o.A);
    println!("o.B {}", o.B);
}
```

Unions allow for different interpretations of underlying bits, fundamentally unsafe. Rust have enum instead.

Type Safety



Bit representation of float 2.0 in IEEE754, is the same as the 32 bit integer `1 << 30 | 1`

Type Safety

```
chibinz@nixos ~/c/h/demo (main)> ./union
input value for option_a: 1
option_a 1
option_b interpreted as 0.000000
chibinz@nixos ~/c/h/demo (main)> ./union
input value for option_a: 1073741825
option_a 1073741825
option_b interpreted as 2.000000
```

Bit representation of float 2.0 in IEEE754, is the same as the 32 bit integer `1 << 30 | 1`

```
error[E0609]: no field `A` on type `Options`
  --> src/union.rs:16:26
   |
16 |     println!("o.A {}", o.A);
   |                         ^ unknown field

error[E0609]: no field `B` on type `Options`
  --> src/union.rs:17:26
   |
17 |     println!("o.B {}", o.B);
   |                         ^ unknown field
```

Rust simply disallows accessing enums by field

Type Safety

```rust
if let Options::A(inner) = o {
    println!("Option::A: {:?}", o);
    println!("Option::B by explict cast: {:?}", Options::B(inner as f32));
}
```

```
input value for Option::A: 2
Option::A: A(2)
Option::B by explict cast: B(2.0)
```

Enums are pattern matched, alternatives must be explicitly constructed
(by casting in this case)

Thread Safety: C

```c
int counter = 0;
void *increment_counter(void *arg) {
    for (int i = 0; i < 100000; i++) {
        counter++; printf("thread %d: %d\n", (int) arg, counter);
    }
}
int main() {
    pthread_t t[8];
    for (int tid = 0; tid < 8; tid++) {
        pthread_create(&t[tid], NULL, increment_counter, (void *) tid);
    }
    for (int tid = 0; tid < 8; tid++) { pthread_join(t[tid], NULL); }
    printf("counter = %d\n", counter);
}
```

Eight threads incrementing a counter in C

Thread Safety: Java

```
public class Race extends Thread {
   public static int counter = 0;
   public static void main(String[] args) {
       var threads = new Thread[8];
       for (int tid = 0; tid < 8; tid++) {
           threads[tid] = new Thread(new Race());
           threads[tid].start();
       }
       for (int tid = 0; tid < 8; tid++) { threads[tid].join(); }
       System.out.println("counter = " + counter);
   }
   public void run() {
       for (int i = 0; i < 100000; i++) { counter++; /* print counter */  }
   }
}
```

Similar implementation in Java, does it sum up to 800000?

Thread Safety

Unfortunately, neither the C nor the Java version works…

Threads are reading and writing concurrently.

Reading an old value of `counter` will cause issues…

Try marking the `run` method directly with `synchronized`, does it work?

```
thread 3: 799914
thread 3: 799915
thread 3: 799916
thread 3: 799917
thread 3: 799918
thread 3: 799919
thread 3: 799920
thread 3: 799921
thread 3: 799922
thread 3: 799923
thread 3: 799924
thread 3: 799925
thread 3: 799926
thread 3: 799927
counter = 799927
```

C

```
counter: 799970
counter: 799971
counter: 799972
counter: 799973
counter: 799974
counter: 799975
counter: 799976
counter: 799977
counter: 799978
counter: 799979
counter: 799980
counter: 799981
counter: 799982
counter: 799983
counter = 799983
```

Java

Thread Safety: Java `synchronized`

`synchronized` utilize **per object instance locks**.

`counter` in this case is a static class variable.

So putting `synchronized` directly before `run` has no effects, we need to lock `Race.class` instead.

Now it works!

```
synchronized(Race.class) {
    counter++;
}
```

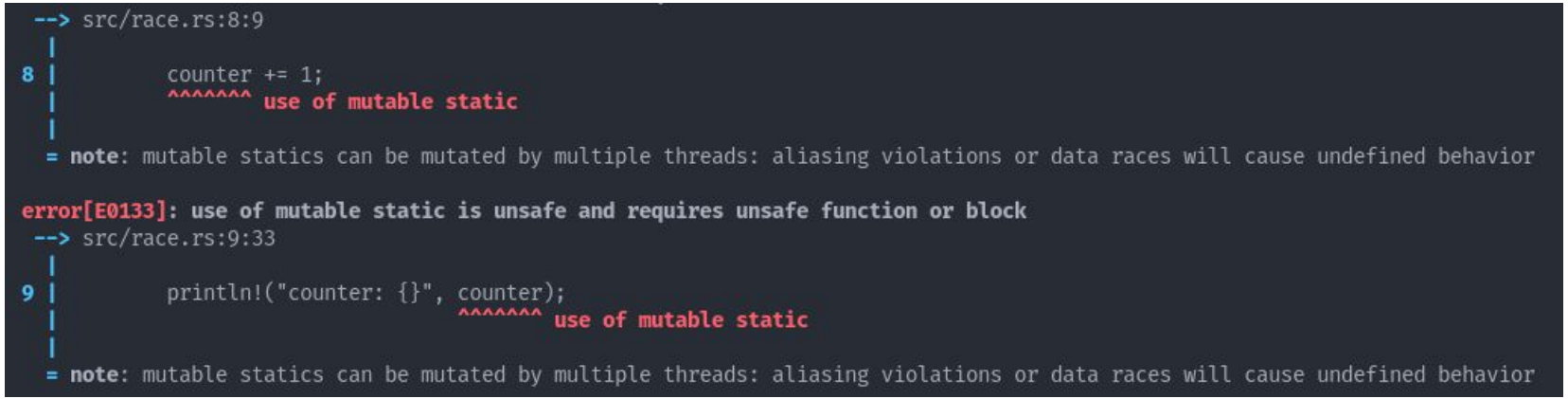
```
counter: 799987
counter: 799988
counter: 799989
counter: 799990
counter: 799991
counter: 799992
counter: 799993
counter: 799994
counter: 799995
counter: 799996
counter: 799997
counter: 799998
counter: 799999
counter: 800000
counter = 800000
```

Thread Safety: Rust

```
static mut counter: i32 = 0;
fn increment_counter() {
    for _  in 0..100000 {
        counter += 1;
        println!("counter: {}", counter);
    }
}
fn main() {
    let mut threads = Vec::new();
    for _ in 0..8 {
        threads.push(thread::spawn(increment_counter));
    }
    for t in threads { t.join().unwrap(); }
    println!("counter = {:?}", counter);
}
```

Let’s try implementing the same logic in Rust… Does it compile?

Thread Safety: Rust



Rust immediately recognizes that using global variables is bad, and points out it could lead to race conditions! Let's try to fix this.

Thread Safety

```rust
use std::sync::atomic::{AtomicI32, Ordering};

static counter: AtomicI32 = AtomicI32::new(0);

fn increment_counter() {
    for _  in 0..100000 {
        counter.fetch_add(1, Ordering::Relaxed);
        println!("counter: {:?}", counter);
    }
}
```

Using atomic variables in Rust solves the issue.

Thread Safety

```
thread 3: 799914
thread 3: 799915
thread 3: 799916
thread 3: 799917
thread 3: 799918
thread 3: 799919
thread 3: 799920
thread 3: 799921
thread 3: 799922
thread 3: 799923
thread 3: 799924
thread 3: 799925
thread 3: 799926
thread 3: 799927
counter = 799927
```

C

```
counter: 799970
counter: 799971
counter: 799972
counter: 799973
counter: 799974
counter: 799975
counter: 799976
counter: 799977
counter: 799978
counter: 799979
counter: 799980
counter: 799981
counter: 799982
counter: 799983
counter = 799983
```

Java, no `synchronized`

```
counter: 799987
counter: 799988
counter: 799989
counter: 799990
counter: 799991
counter: 799992
counter: 799993
counter: 799994
counter: 799995
counter: 799996
counter: 799997
counter: 799998
counter: 799999
counter: 800000
counter = 800000
```

Rust, correct result!