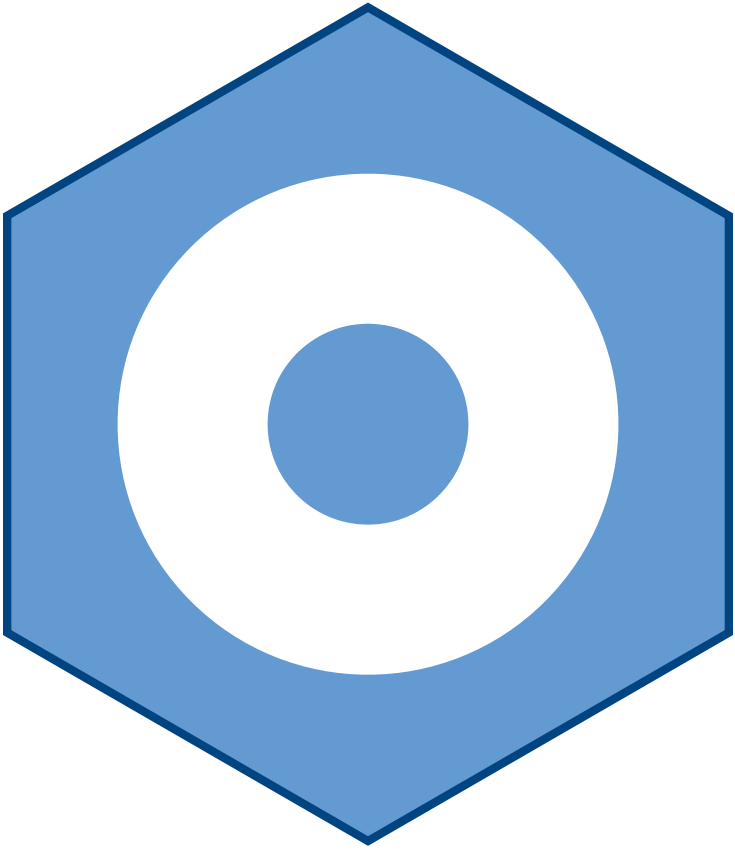
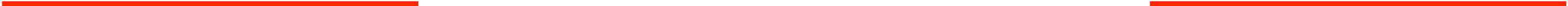
50

```
void entry_point (
        shared_ptr<mutex<string>> data,
        int thread_id) safe
{
    auto lock_guard = data->lock();

    string^s = lock_guard^.borrow();
    s^->append ("🔥");

    println (*s);
}

int main() safe
{
    //...
    threads^.push_back(thread (&entry_point,
                               copy shared_data, i));
}
```

```
void entry_point (
        std::shared_ptr<synchronized_value<std::string>> data,
        int tid)
{
    apply ([tid] (auto& s) {
        s.append ("🔥");
        std::println ("{} {}", s, tid);
        return s;
    },
    *data);
}

int main()
{
    //...
    threads.push_back (safe_thread (entry_point,
                                    auto (s), auto (i)));
}
```





```
void entry_point (
        std::shared_ptr<synchronized_value<std::string>> data,
        int tid)
{
    apply ([tid] (auto& s) {
        s.append ("🔥");
        std::println ("{} {}", s, tid);
        return s;
    },
    *data);
}

int main()
{
    //...
    threads.push_back (safe_thread (entry_point,
                                    auto (s), auto (i)));
}
```

```
void entry_point (
        shared_ptr<mutex<string>> data,
        int thread_id) safe
{
    auto lock_guard = data->lock();

    string^s = lock_guard^.borrow();
    s^->append ("🔥");

    println (*s);
}

int main() safe
{
    //...
    threads^.push_back(thread (&entry_point,
                               copy shared_data, i));
}
```

Same example in Rust

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn entry_point(data: Arc<Mutex<String>>, thread_id: i32) {
    let mut guard = data.lock().unwrap();
    guard.push_str("🔥");
    println!("Thread {}: {}", thread_id, *guard);
}

pub fn main() {
    let shared_data = Arc::new(Mutex::new(String::from("Hello threads")));

    let mut threads = Vec::new();

    const NUM_THREADS: i32 = 15;

    for i in 0..NUM_THREADS {
        // Clone the Arc for this thread
        let data_clone = Arc::clone(&shared_data);

        // Spawn the thread and store its handle
        let handle = thread::spawn(move || {
            entry_point(data_clone, i);
        });

        threads.push(handle);
    }

    for handle in threads {
        handle.join().unwrap();
    }
}
```