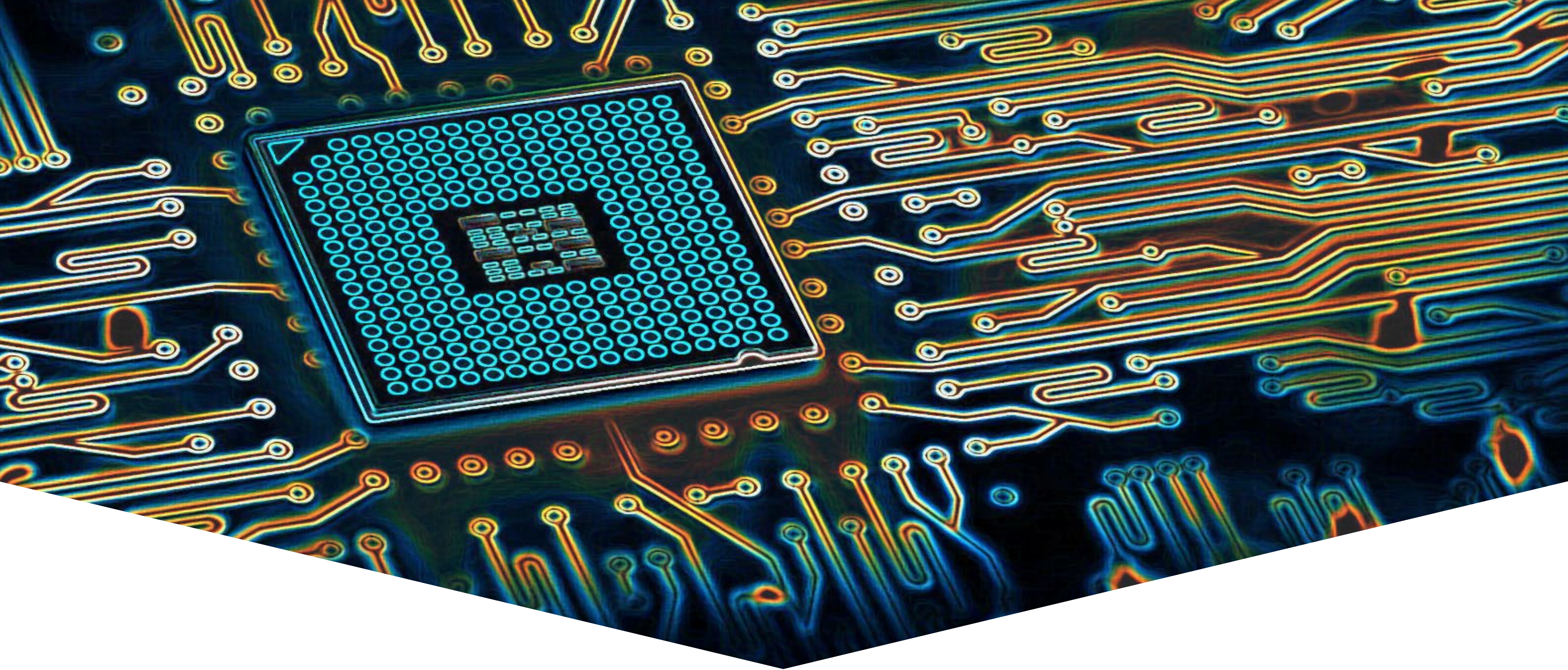
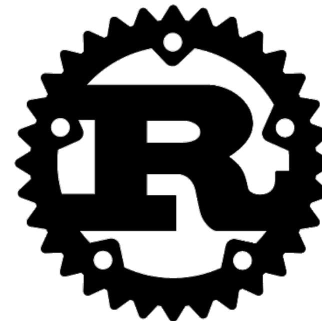


RTS10 Week 7

Leerdoelen Week 7 en 8

Leerdoelen week 7 en 8. Je leert:

- Hoe je Rust kunt gebruiken als alternatief voor C/C++ als systems programming language;
- De belangrijkste conceptuele verschillen tussen Rust en C/C++.
- Rust kunt gebruiken om een microcontroller te programmeren.



The Rust Programming Language

HOGESCHOOL ROTTERDAM

Disclaimer

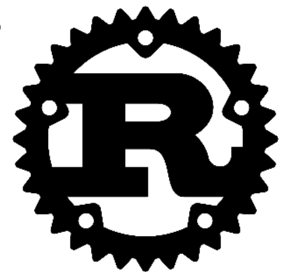
- Dit is **geen** Rust programmeercursus.
  - Rust is een uitgebreide taal, met een uitgebreide standaard library (vergelijkbaar met C++).
- Docenten zijn enthousiast maar zeker nog **geen** experts.
  - Rust heeft een vrij steile leercurve (zeker als je C gewend bent), ook voor docenten.
  - Tips en trucs zijn welkom!

HOGESCHOOL ROTTERDAM

C versus Rust

- Je kan het goed doen
  - Zo niet: run time error, segmentation fault, core dumped
  - Debuggen heel lastig
- Je kan het niet verkeerd doen
  - Probeer je het toch: compile time error
  - Compiler geeft suggesties en kan deze op verzoek doorvoeren.

Bijvoorbeeld: NULL pointer, memory leak (free vergeten), memory corruption (free teveel), dangling reference (pointer naar geheugen dat al vrijgegeven is), race condities bij multithreaded code (gedeelde resource niet beveiligd, unlock vergeten).



HOGESCHOOL ROTTERDAM

C is niet memory safe

```c
const char* eerste_klinker(const char* s) {
    while (*s != '\0') {
        if (*s == 'a' || *s == 'e' || *s == 'i' || *s == 'o' || *s == 'u') {
            return s;
        }
        s++;
    }
    return NULL;
}

int main(void) {
    const char* woord = "hallo";
    const char* klinker = eerste_klinker(woord);
    printf("De eerste klinker in %s is %c\n", woord, *klinker);
```

HOGESCHOOL ROTTERDAM

C is niet memory safe

```c
const char* eerste_klinker(const char* s) {
    while (*s != '\0') {
        if (*s == 'a' || *s == 'e' || *s == 'i' || *s == 'o' || *s == 'u') {
            return s;
        }
        s++;
    }
    return NULL;
}

int main(void) {
    const char* woord = "hallo";
    const char* klinker = eerste_klinker(woord);
    if (klinker != NULL) {
        printf("De eerste klinker in %s is %c\n", woord, *klinker);
    }
    else {
        printf("Er is geen klinker in %s\n", woord);
    }
```

HOGESCHOOL ROTTERDAM

C is niet memory safe

```c
const char* eerste_klinker(const char* s) {
    while (*s != '\0') {
        if (*s == 'a' || *s == 'e' || *s == 'i' || *s == 'o' || *s == 'u') {
            return s;
        }
        s++;
    }
    return NULL;
}

int main(void) {
    const char* woord = "hallo";
    char* heap_woord = malloc((strlen(woord) + 1) * sizeof(char));
    strcpy(heap_woord, woord);
    // heap_woord = NULL;
    // free(heap_woord);
    klinker = eerste_klinker(heap_woord);
```

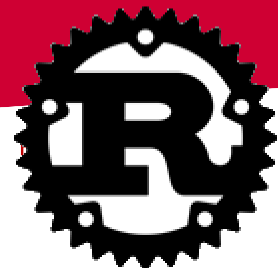

HOGESCHOOL ROTTERDAM

Rust is memory safe

```
fn eerste_klinker(s: &str) -> Option<char> {
    for c in s.chars() {
        if c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' {
            return Some(c);
        }
    }
    None
}

fn main() {
    let woord = "hallo";
    let klinker = eerste_klinker(woord);
    // println!("De eerste klinker in {} is {}", woord, klinker); // compile error
    match klinker {
        Some(klinker) => println!("De eerste klinker in {} is {}", woord, klinker),
        None => println!("Er is geen klinker in {}", woord),
    }
    // None vergeten: compile error: pattern `None` not covered
```


HOGESCHOOL ROTTERDAM

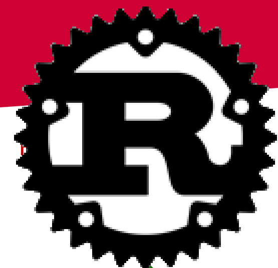
Rust is memory safe



```
fn main() {
    // let klinker = eerste_klinker(None);
    // compile error: expected `&str`, found enum `Option`

    let heap_word = String::from("hallo");
    drop(heap_word); // heap_word is nu niet meer beschikbaar
    // compile error op volgende regel: value borrowed here after move
    let klinker = eerste_klinker(&heap_word);
    match klinker {
        Some(klinker) => println!("De eerste klinker in {} is {}", woord, klinker),
        None => println!("Er is geen klinker in {}", woord),
    }
}
```


HOGESCHOOL ROTTERDAM

Rust is memory safe



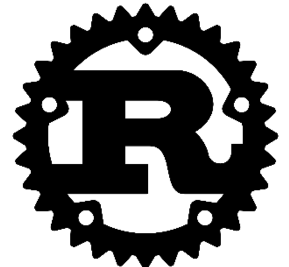
```
// Er kan maar één eigenaar (owner) zijn van een waarde op de heap
// - Niet mogelijk om free te vergeten (automatisch als owner out of scope gaat)
// - Geen double-free error mogelijk!
// Een pointer/reference (borrow) kan niet langer bestaan dan de eigenaar
// - Geen dangling pointer/reference (use after free) error mogelijk!

fn main() {
    let mut x = vec![11, 12, 13];
    // let p = &x[3];
    // 'main' panicked at 'index out of bounds: the len is 3 but the index is 3'
    x.push(14);
    let p = &x[3];
    println!("*p = {}", *p);
    let mut y = x; // move i.p.v. copy: y is nu de eigenaar, x bestaat niet meer
    // println!("{}", *p); // error^: cannot move out of `x` because it is borrowed
    // x.push(15); // error: use of moved value: `x`
    y.push(15);
    println!("{:?}", y);
}
```

C versus Rust

- Snel
  - Variabelen zijn default read write
  - Keyword `const` als je read only wilt

- Minstens zo snel
  - Variabelen zijn default read only
  - Keyword `mut` voor als je read write wilt

Als je weet dat een variabele read only is kan de compiler agressiever optimaliseren, dus het is een gemiste kans als de programmeur dat vergeet te vermelden.



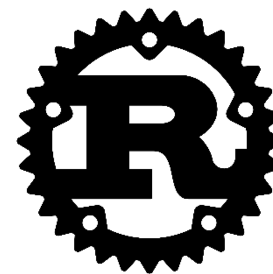
HOGESCHOOL ROTTERDAM

C versus Rust

- Snel
  - Pointer parameters kunnen `restrict` (keyword, sinds C99) gemaakt worden (er kan geen andere pointer naar dezelfde data wijzen).

- Minstens zo snel
  - Mutable pointers zijn altijd restricted.
  - Voorkomt data races (synchronisatieproblemen in multithreaded code).

Als je weet dat pointers restricted zijn kan de compiler agressiever optimaliseren, dus het is een gemiste kans als de programmeur dat vergeet te vermelden.

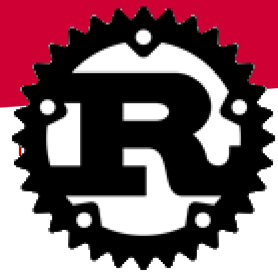


C restrict

```
void inc_first_add_to_second(int *restrict x, int *restrict y) {
    *x += 1;
    *y += *x;
}

int main() {
    int i = 0;
    int j = 0;
    inc_first_add_to_second(&i, &j);
    printf("i: %d, j: %d\n", i, j);
    inc_first_add_to_second(&i, &i);
    printf("i: %d, j: %d\n", i, j);
    // $ gcc -O0 restrict.c && ./a.out
    // i: 1, j: 1
    // i: 4, j: 1
    // $ gcc -Os restrict.c && ./a.exe
    //  i: 1, j: 1
    //  i: 2, j: 1
    return 0;
```


HOGESCHOOL ROTTERDAM

Rust borrow checker

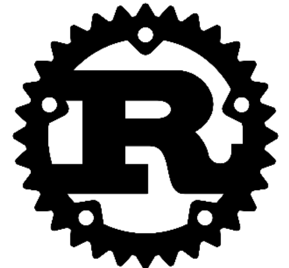


```rust
fn inc_first_add_to_second(x: &mut i32, y: &mut i32) {
    *x += 1;
    *y += *x;
}

fn main() {
    let mut i = 0;
    let mut j = 0;
    inc_first_add_to_second(&mut i, &mut j);
    println!("i: {}, j: {}\n", i, j);
    // inc_first_add_to_second(&mut i, &mut i);
    // error: cannot borrow `i` as mutable more than once at a time
}
```

C versus Rust

- Error waarde die functie teruggeeft wordt default genegeerd

- Error waarde die functie teruggeeft moet afgehandeld (mogelijk genegeerd) worden.



C error afhandling

```c
void *print1(void *par) {
    for (int i = 0; i < 10; i++) { usleep(100000); printf("print1\n"); }
    return NULL;
}

void *print2(void *par) {
    for (int i = 0; i < 10; i++) { usleep(200000); printf("print2\n"); }
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &print1, NULL);
    pthread_create(&t2, NULL, &print2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t1, NULL);
    return EXIT_SUCCESS;
}
```

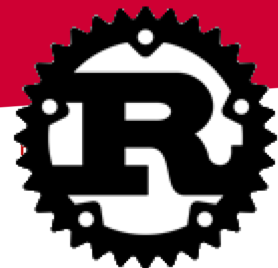
HOGESCHOOL ROTTERDAM

Betere C error afhandling

```
void check(int error)
{
    if (error != 0)
    {
        printf("Error: %s\n", strerror(error));
        exit(EXIT_FAILURE);
    }
}

// ...
int main(void) {
    pthread_t t1, t2;
    check( pthread_create(&t1, NULL, &print1, NULL) );
    check( pthread_create(&t2, NULL, &print2, NULL) );
    check( pthread_join(t1, NULL) );
    check( pthread_join(t1, NULL) );
    return EXIT_SUCCESS;
}
```


HOGESCHOOL ROTTERDAM

Rust error afhandeling



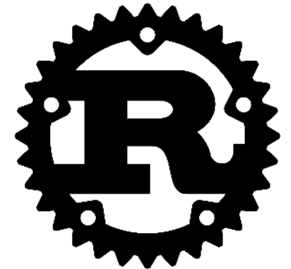
```rust
fn print1() { /* ... */ }
fn print2() { /* ... */ }

fn main() {
    let t1 = thread::spawn(print1);
    let t2 = thread::spawn(print2);
    // t1.join();
    // warning: this `Result` may be an `Err` variant, which should be handled
    match t1.join() {
        Ok(_) => println!("t1 finished"),
        Err(e) => println!("t1 failed: {:?}", e),
    }
    // t1.join().unwrap(); // panics on failure
    // t1.join().expect("t1 failed"); // panics on failure with message

    // t1.join().unwrap();
    // error: use of moved value: `t1`
    t2.join().unwrap();
}
```


HOGESCHOOL ROTTERDAM

C versus Rust

- Losse tools:
  - build (make),
  - test framework (Cunit, Catch2),
  - static analyzer (Cppcheck),
  - documentation (doxygen), …

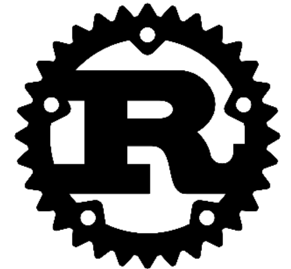
- Alles inclusief:
  - cargo new,
  - cargo build,
  - cargo test,
  - cargo check,
  - cargo fix,
  - cargo doc,
  - …



C versus Rust

- Eenvoudig
  - Foutgevoelig

- Uitgebreid
  - Compiler voorkomt fouten



HOGESCHOOL ROTTERDAM

C print array

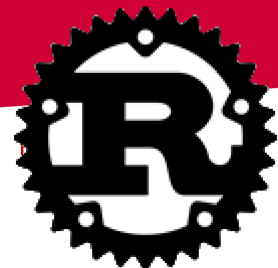
```c
void print_array(int array[], size_t size) {
    for (size_t i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main(void) {
    int a[] = {1, 2, 3, 4, 5};
    print_array(a, 5);
    return 0;
}
```

Wat kan er fout gaan:

- array is `NULL` of invalid (use after free)
- `size` is groter dan de grootte van de array
- logische fouten in `for` -loop:
  - 1 i.p.v. `0`
  - <= i.p.v. <
  - `i` aanpassen in loop

HOGESCHOOL ROTTERDAM

Rust print array



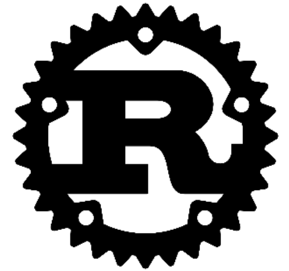
```
fn main() {
    let a = [1, 2, 3, 4, 5];
    print_array(&a);
}

fn print_array(array: &[i32]) {
    for i in array {
        print!("{} ", i);
    }
    println!();
}

fn print_array(array: &[i32]) {
    array.iter().for_each(|i| print!("{} ", i));
    println!();
}
```


HOGESCHOOL ROTTERDAM

C versus Rust

- Concurency is moeilijk
  - Kans op problemen bij threads met gemeenschappelijk geheugen / peripheral

- Fearless concurency
  - Geen problemen tijdens run-time

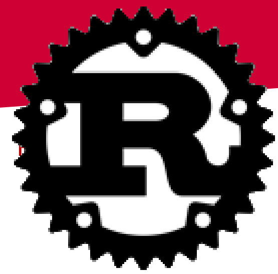


C pthread mutex

```
int aantal = 0;
pthread_mutex_t m;

void *teller(void *par) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&m); aantal++; pthread_mutex_unlock(&m);
    }
    return NULL;
}
int main(void)
{
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &teller, NULL);
    aantal++;
    pthread_create(&t2, NULL, &teller, NULL);
    check( pthread_join(t1, NULL) );
    check( pthread_join(t2, NULL) );
    printf("aantal = %d\n", aantal);
```

HOGESCHOOL ROTTERDAM

Rust mutex



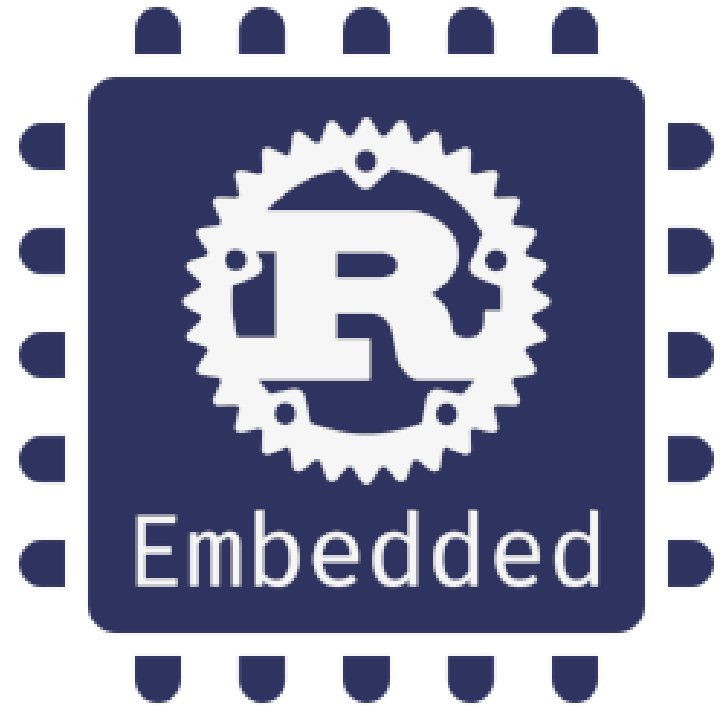
```rust
use std::sync::{Mutex, Arc};

fn teller(m: Arc<Mutex<u32>>) {
    for _i in 1..=100000 {
        let mut aantal = m.lock().unwrap();
        *aantal += 1;
    }
}

fn main() {
    let a = Arc::new(Mutex::new(0));
    let m = a.clone();
    let t1 = thread::spawn(||{teller(m)});
    let m = a.clone();
    let t2 = thread::spawn(||{teller(m)});
    t1.join().unwrap();
    t2.join().unwrap();
    let aantal = a.lock().unwrap();
    println!("aantal = {}", aantal);
```


HOGESCHOOL ROTTERDAM

Volgende week…

Embedded Rust.

- In week 8 geen les op school maar alleen de online les.
- Korte introductie bij de start van deze online les.



Aan de slag!

Aan de slag met Opdrachten_Week_7.pdf