Bezpieczeństwo języka Rust

Czyli lekcja o tym jak zabezpieczać się przed niechcianymi wpadkami.

Kasper Seweryn

Null Safety

- **Rust:** Brak wartości null
- **Inne języki:** często dopuszczają wartości null, co prowadzi do błędów

Null Safety (c.d.)

Co w takim razie zrobić, gdy potrzebujemy wartości, która może być pusta?

```
enum Option<T> {
  Some(T),
  None,
}
```

```
let x: Option<i32> = Some(5);
let y: Option<i32> = None;
```

```
let z = y.unwrap(); // Panikuje, gdy y jest równe None
```

```
match x {
  Some(5) => println!("x is 5"),
  Some(value) => println!("x is {}", value),
  None => println!("x has no value"),
}
```

Obsługa błędów

```
enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

```
fn write_to_file() -> Result<(), io::Error> {
  let mut f = File::create("hello.txt")?;
  f.write_all(b"Hello, world!")?;
  Ok(())
}
```

```
fn main() {
  match write_to_file() {
    Err(e) => println!("Error writing file: {}", e),
    _ => {} // Ewentualnie: Ok(()) => {}
  }
}
```

Ownership i Borrowing

- Koncept własności (ownership)
- Koncept pożyczania (borrowing)
- Brak mechanizmu Garbage Collector

Ownership

Przekazywanie własności

```
fn print_length(s: String) {
    println!("Length: {}", s.len());
}
```

```
fn main() {
  let s = "hello".to_string();
  print_length(s); // Przekazujemy własność s do funkcji print_length
                   // Pod koniec wykonywania funkcji, zmienna s jest usuwana
  print_length(s); // Błąd kompilacji: Nie mamy już dostępu do zmiennej s
}
```

Borrowing

Pożyczanie zmiennych

```
fn print_length(s: &String) {
    println!("Length: {}", s.len());
}
```

```
fn main() {
  let s = "hello".to_string();
  print_length(&s); // Pożyczamy zmienną s do funkcji
  print_length(&s); // Pożyczamy zmienną s do funkcji, brak błędu kompilacji
  // Właścicielem s dalej jest main
  // Więc wciąż mamy dostęp do zmiennej s
}
```

Zmienialne pożyczanie

```cpp
// C++
std::string s1 = "hello";
print_length(s1);
```

```cpp
void print_length(std::string& s1) {
    s1 += "!";
    std::cout << "Length of string s1: " << s1.length() << std::endl;
}
```

```rust
// Rust
fn print_length(s: &mut String) {
    println!("Length: {}", s.len());
}
```

```rust
let mut s = "hello".to_string();
print_length(&mut s);
```

Jakie błędy zostały wyeliminowane przez te koncepty?

- Null Pointer Dereference
- Memory Leak
- Dangling Pointer
- Use After Free
- Double Free

Dlaczego to ważne? Bo błędy związane z pamięcią to:

- 49% podatności w Chrome
- 72% podatności w Firefoxie
- 88% podatności w jądrze macOS
- 70% podatności w produktach Microsoftu
- 65% podatności w Androidzie
- 65% podatności w jądrach Ubuntu

Źródło: https://static.sched.com/hosted_files/lssna19/d6/kernel-modules-in-rust-lssna2019.pdf

Ale przecież oddajemy kompilatorowi pełną kontrolę nad pamięcią!

I tu wkraczają inteligentne wskaźniki (smart pointers)

- **Box<T>** - wskaźnik na alokowaną na stercie (heap) wartość typu T. Podstawowo zmienne są alokowane na stosie (stack). Daje nam możliwość samodzielnego zwolnienia pamięci przed tym jak zmienna wyjdzie ze scope'a.

- **Cell<T>** - wskaźnik na wartość typu T, który pozwala na mutowanie wartości, nawet gdy wskaźnik jest niezmienialny (immutable). Zapewnia mutację poprzez kopiowanie wartości bez narzutu czasowego.

- **RefCell<T>** - wskaźnik na wartość typu T, który pozwala na dynamiczne pożyczanie podczas wykonywania kodu. Pozwala na wiele zmienialnych oraz niezmienialnych pożyczeń na raz. Sprawdza za pomocą mechanizmu borrow checker, czy pożyczenia są poprawne podczas wykonywania kodu, co negatywnie wpływa na wydajność czasową. Można go stosować do typów i struktur niekopiowalnych.

- **Rc<T>** - wskaźnik na wartość typu T, która może mieć wiele właścicieli. Licznik referencji jest zwiększany, gdy wskaźnik jest kopiowany i zmniejszany, gdy wskaźnik jest usuwany. Wartość jest usuwana z pamięci, gdy licznik referencji spadnie do zera.

- **Arc<T>** - to samo co Rc, ale zaimplementowane w sposób bezpieczny dla wielu wątków.

Bezpieczna wielowątkowość

Poprzednio opisane mechanizmy takie jak ownership oraz borrowing pozwalają na bezpieczne programowanie wielowątkowe, lecz nie są to jedyne mechanizmy.

Atomiczność

- **AtomicBool** - typ bool, który może być bezpiecznie używany przez wiele wątków jednocześnie. Wartość jest atomowo zmieniana, co oznacza, że nie może zostać przerwana przez inny wątek.
- **AtomicUsize** - typ liczbowy, który może być bezpiecznie używany przez wiele wątków jednocześnie. Wartość jest atomowo zmieniana, co oznacza, że nie może zostać przerwana przez inny wątek.

Synchronizacja

- **Mutex<T>** - wskaźnik na wartość typu T, który pozwala na mutowanie wartości, nawet gdy wskaźnik jest niezmienialny (immutable). Zapewnia mutację poprzez blokowanie innych wątków, gdy wskaźnik jest używany przez jeden wątek.
- **RwLock<T>** - wskaźnik na wartość typu T, który pozwala na wiele niezmienialnych pożyczeń albo jedno zmienialne pożyczenie na raz. Blokuje dostęp dla innych wątków zarówno do odczytu jak i zapisu, gdy wskaźnik jest używany do zapisu przez jeden wątek.

Co jeżeli potrzebujemy jeszcze więcej władzy?

Kilka słów o klauzuli `unsafe`.

- Gdy potrzebujemy większej kontroli nad pamięcią, możemy wykorzystać blok `unsafe`.

```
let mut data = vec![1, 2, 3];

unsafe {
    // Pobieramy surowy wskaźnik
    let ptr = data.as_mut_ptr();

    // Zmieniamy wartość pod indeksem 1 używając surowego wskaźnika
    *ptr.offset(1) = 5;
}
```

- Funkcje oznaczone jako `unsafe` mogą być wywoływane tylko w blokach `unsafe`.

```
unsafe fn dangerous() { /* ... */ }
```

- Gdy używamy `unsafe`, to sami jesteśmy odpowiedzialni za zapewnienie bezpieczeństwa.
- Gwarantuje to, że niebezpieczny kod nie będzie wykonywany nieświadomie.

Rozszerzanie bezpieczeństwa gwarantowanego przez kompilator

Porozmawiajmy chwilę o języku SQL

- Język rust możemy rozszerzać za pomocą makr, które generują kod na etapie kompilacji. Makra możemy rozpoznać po znaku wykrzyknika `!` na końcu nazwy funkcji.
- Biblioteka `sqlx` dostarcza makro `sqlx::query!`, które pozwala na bezpieczne wykonywanie zapytań SQL na bazie danych.
- Podczas kompilacji, makro `sqlx::query!` sprawdza poprawność zapytania SQL poprzez wykonanie odpowiedniego zapytania na bazie danych przeznaczonej do testów.
- Jeśli zapytanie jest niepoprawne lub zwraca inne typy niż oczekiwane przez Rust, to kompilacja zostanie przerwana z odpowiednim błędem.

Podsumowanie

Rust zapewnia bezpieczeństwo głównie przez wszechwiedzący kompilator, który dokonuje wszechstronnej analizy kodu

- Wykrywa błędy podczas kompilacji, zanim kod zostanie uruchomiony
- Wykrywa niezainicjalizowane zmienne
- Wykrywa nieużywane zmienne
- Wykrywa błędy wypożyczania (np: podwójne mutowalne pożyczenie)
- Wykrywa problemy w zarządzaniu pamięcią
- Wykrywa nieobsłużone błędy zwracane z funkcji (`` `Result<T, E>` ``)
- Wykrywa nieobsłużone puste wartości zmiennych (`` `Option<T>` ``)
- I wiele innych…

Polecana literatura

- The Rust Book - oficjalna książka o języku Rust
- Rust by Example - przykłady kodu w języku Rust
- The Rustonomicon - zaawansowane zagadnienia w języku Rust
- The Unstable Book - dokumentacja eksperymentalnych funkcji języka Rust

Dziękuję za uwagę.