Mechanizmy bezpieczeństwa Rust z perspektywy C++

O języku Rust słyszał już chyba każdy programista. Nie znaczy to oczywiście, że wszyscy potrafią z niego korzystać. Faktycznie jednak zyskał on niezwykłą popularność w bardzo krótkim czasie, a to za sprawą unikatowej cechy, jaką jest gwarancja bezpieczeństwa pamięci. Mowa tutaj zarówno o bezpieczeństwie pod kątem wycieków pamięci, jak i wielowątkowego dostępu do niej. W artykule przyjrzymy się mechanizmom języka Rust, temu, jak one działają, a także czy i w jaki sposób możemy wykorzystać je w języku C++.

TROCHĘ HISTORII

Początek Rust sięga 2006 roku, kiedy to Greydon Hoare, pracownik firmy Mozilla, rozpoczął pracę nad tym językiem. W miarę postępów w implementacji Mozilla przejęła pieczę nad projektem i sponsorowała go. W 2012 roku pojawiła się pierwsza wersja alpha tego języka, a w 2015 usłyszeliśmy o jego pierwszej stabilnej wersji.

Po uzyskaniu popularności i po tym, jak został wykorzystany już w kilku projektach (a nawet zaproponowano, żeby nowe moduły jądra Linux pisać w tym języku), utworzono Fundację Rust, odpowiedzialną za rozwój tego projektu. W jej skład wchodzą takie firmy jak AWS, Google, Microsoft i oczywiście sama Mozilla.

Na marginesie, warto wspomnieć, że obecnie po raz szósty z rzędu Rust jest najbardziej lubianym przez programistów językiem według ankiety Stack Overflow[1].

Nasuwa się jednak pytanie, po co kolejny język, komu ma on służyć i jakie problemy rozwiązywać? W zasadzie Rust miał być bardzo podobny do C++, ponieważ podstawowe założenia tego projektu są zbieżne. Między innymi miał to być język do programowania systemowego, czyli musiał być kompilowalny do kodu maszynowego, i podobnie jak C++ miał być bezpieczny i przyjazny dla programowania wielowątkowego.

Przyjrzyjmy się temu zagadnieniu bliżej.

WŁAŚCIWOŚCI JĘZYKA RUST

Rust jako nowoczesny język ma wszystko, co jest potrzebne do szybkiego tworzenia aplikacji. Możemy się więc spodziewać:

- wbudowanego menadżera pakietów (którym jest Cargo),
- programowania generycznego (nieco podobnego do szablonów C++),
- wbudowanego frameworku do testowania,
- modułów,
- wyrażeń dopasowanych do wzorca (ang. *pattern matching expressions*).

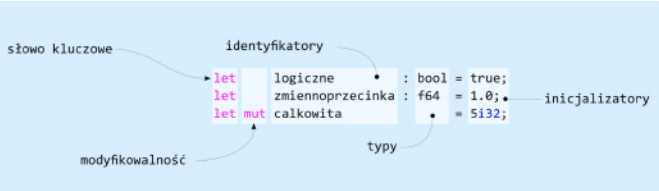
Co więcej, język Rust oferuje wspomniane już bezpieczeństwo pamięci, a co ciekawe, bezpieczeństwo to jest zapewnione bez użycia odśmiecacza pamięci (ang. *Garbage Collector*). W języku tym znajdziemy również mechanizmy typowe dla języka C++, takie jak chociażby RAII.

1. https://insights.stackoverflow.com/survey/2021#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages.

Zanim przejdziemy do omawiania dalszych właściwości Rust, musimy najpierw zapoznać się z podstawami jego składni, aby nie mieć problemów ze zrozumieniem przykładów. Jeżeli podstawy tego języka są już czytelnikowi znane, może pominąć kolejny rozdział i przejść od razu do sekcji „Mechanizmy bezpieczeństwa w Rust".

Zmienne w języku Rust

Składnię deklaracji zmiennych najlepiej zobrazuje nam Rysunek 1.



Rysunek 1. Składnia deklaracji zmiennych w Rust

Jak widać, każda deklaracja zmiennej musi rozpocząć się słowem kluczowym `let`. W języku Rust domyślnie wszystkie zmienne są niemodyfikowalne i aby to zmienić, należy zaraz po słowie `let` umieścić słowo `mut`. Sprawi to, że kompilator będzie pozwalał na modyfikację takich zmiennych.

Po słowach kluczowych natomiast następuje określenie identyfikatora zmiennej, czyli jej nazwy.

Podobnie jak w języku C++, typ zmiennej może być określony wprost lub wydedukowany. Jeżeli jawnie chcemy określić typ zmiennej, podajemy go po identyfikatorze zgodnie z przykładami widocznymi na Rysunku 1.

Zmienne mogą być inicjalizowane zaraz przy ich deklaracji lub inicjalizowane z opóźnieniem, podobnie jak w C++, jednak jeżeli spróbujemy użyć takiej zmiennej w Rust przed jej inicjalizacją, dostaniemy błąd kompilacji.

Kolejną różnicą względem języka C++ jest to, że dedukcja typu zmiennej w języku Rust może być opóźniona. Na przykład poprawny jest fragment kodu przedstawiony w Listingu 1.

Listing 1. Leniwa dedukcja typu

```
let wektor; // wektor liczb całkowitych
let wektor2; // wektor łańcuchów znaków

wektor = vec![1, 2, 3];
wektor2 = vec!["1", "2", "3"];
```