Primitivos

O Rust fornece acesso a uma ampla variedade de `primitives` . Um exemplo inclui:

Tipos escalares

- Inteiros assinados: `i8` , `i16` , `i32` , `i64` , `i128` e `isize` (tamanho do ponteiro)
- Inteiros sem sinal: `u8` , `u16` , `u32` , `u64` , `u128` e `usize` (tamanho do ponteiro)
- Ponto flutuante: `f32` , `f64`
- `char` Valores escalares Unicode como `'a'` , `'α'` e `'∞'` (4 bytes cada)
- `bool` ou `true` ou `false`
- O tipo de unidade `()` , cujo único valor possível é uma tupla vazia: `()`

Apesar do valor de um tipo de unidade ser uma tupla, ele não é considerado um tipo composto porque não contém múltiplos valores.

Tipos de compostos

- Matrizes como `[1, 2, 3]`
- Tuplas como `(1, true)`

Variáveis sempre podem ser *anotadas por tipo* . Números também podem ser anotados por meio de um *sufixo* ou *por padrão* . Inteiros são anotados por padrão `i32` e floats são anotados por padrão `f64` . Observe que Rust também pode inferir tipos a partir do contexto.

```rust
fn main() {
    // Variables can be type annotated.
    let logical: bool = true;

    let a_float: f64 = 1.0;  // Regular annotation
    let an_integer   = 5i32; // Suffix annotation

    // Or a default will be used.
    let default_float   = 3.0; // `f64`
    let default_integer = 7;   // `i32`

    // A type can also be inferred from context.
    let mut inferred_type = 12; // Type i64 is inferred from another line.
    inferred_type = 4294967296i64;

    // A mutable variable's value can be changed.
    let mut mutable = 12; // Mutable `i32`
    mutable = 21;

    // Error! The type of a variable can't be changed.
    mutable = true;

    // Variables can be overwritten with shadowing.
    let mutable = true;

    /* Compound types - Array and Tuple */

    // Array signature consists of Type T and length as [T; length].
    let my_array: [i32; 5] = [1, 2, 3, 4, 5];

    // Tuple is a collection of values of different types
    // and is constructed using parentheses ().
    let my_tuple = (5u32, 1u8, true, -5.04f32);
}
```

Veja também:

a `std` biblioteca , `mut` , `inference` , e `shadowing`

Literais e operadores

Inteiros `1`, floats `1.2`, caracteres `'a'`, strings `"abc"`, booleanos `true` e o tipo de unidade `()` podem ser expressos usando literais.

Os inteiros podem, alternativamente, ser expressos usando notação hexadecimal, octal ou binária usando estes prefixos respectivamente: `0x`, `0o` ou `0b`.

Sublinhados podem ser inseridos em literais numéricos para melhorar a legibilidade, por exemplo, `1_000` é o mesmo que `1000`, e `0.000_001` é o mesmo que `0.000001`.

Rust também suporta notação científica E, por exemplo `1e6`, `7.6e-4`. O tipo associado é `f64`.

Precisamos informar ao compilador o tipo dos literais que usamos. Por enquanto, usaremos o `u32` sufixo para indicar que o literal é um inteiro de 32 bits sem sinal e o `i32` sufixo para indicar que é um inteiro de 32 bits com sinal.

Os operadores disponíveis e suas precedências em Rust são semelhantes a outras linguagens do tipo C.

```
fn main() {
    // Integer addition
    println!("1 + 2 = {}", 1u32 + 2);

    // Integer subtraction
    println!("1 - 2 = {}", 1i32 - 2);
    // TODO ^ Try changing `1i32` to `1u32` to see why the type is important

    // Scientific notation
    println!("1e4 is {}, -2.5e-3 is {}", 1e4, -2.5e-3);

    // Short-circuiting boolean logic
    println!("true AND false is {}", true && false);
    println!("true OR false is {}", true || false);
    println!("NOT true is {}", !true);

    // Bitwise operations
    println!("0011 AND 0101 is {:04b}", 0b0011u32 & 0b0101);
    println!("0011 OR 0101 is {:04b}", 0b0011u32 | 0b0101);
    println!("0011 XOR 0101 is {:04b}", 0b0011u32 ^ 0b0101);
    println!("1 << 5 is {}", 1u32 << 5);
    println!("0x80 >> 2 is 0x{:x}", 0x80u32 >> 2);

    // Use underscores to improve readability!
    println!("One million is written as {}", 1_000_000u32);
}
```

Tuplas

Uma tupla é uma coleção de valores de diferentes tipos. Tuplas são construídas usando parênteses `()`, e cada tupla em si é um valor com assinatura de tipo `(T1, T2, ...)`, onde `T1`, `T2` são os tipos de seus membros. Funções podem usar tuplas para retornar múltiplos valores, pois tuplas podem conter qualquer número de valores.

```rust
// Tuples can be used as function arguments and as return values.
fn reverse(pair: (i32, bool)) -> (bool, i32) {
    // `let` can be used to bind the members of a tuple to variables.
    let (int_param, bool_param) = pair;

    (bool_param, int_param)
}

// The following struct is for the activity.
#[derive(Debug)]
struct Matrix(f32, f32, f32, f32);

fn main() {
    // A tuple with a bunch of different types.
    let long_tuple = (1u8, 2u16, 3u32, 4u64,
                      -1i8, -2i16, -3i32, -4i64,
                      0.1f32, 0.2f64,
                      'a', true);

    // Values can be extracted from the tuple using tuple indexing.
    println!("Long tuple first value: {}", long_tuple.0);
    println!("Long tuple second value: {}", long_tuple.1);

    // Tuples can be tuple members.
    let tuple_of_tuples = ((1u8, 2u16, 2u32), (4u64, -1i8), -2i16);

    // Tuples are printable.
    println!("tuple of tuples: {:?}", tuple_of_tuples);

    // But long Tuples (more than 12 elements) cannot be printed.
    //let too_long_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13);
    //println!("Too long tuple: {:?}", too_long_tuple);
    // TODO ^ Uncomment the above 2 lines to see the compiler error

    let pair = (1, true);
    println!("Pair is {:?}", pair);

    println!("The reversed pair is {:?}", reverse(pair));

    // To create one element tuples, the comma is required to tell them apa
    // from a literal surrounded by parentheses.
    println!("One element tuple: {:?}", (5u32,));
    println!("Just an integer: {:?}", (5u32));

    // Tuples can be destructured to create bindings.
    let tuple = (1, "hello", 4.5, true);

    let (a, b, c, d) = tuple;
    println!("{:?}, {:?}, {:?}, {:?}", a, b, c, d);

    let matrix = Matrix(1.1, 1.2, 2.1, 2.2);
    println!("{:?}", matrix);
}
```

Atividade

1. *Recapitulação* : adicione a `fmt::Display` característica à `Matrix` estrutura no exemplo acima, de modo que, se você alternar da impressão do formato de depuração `{:?}` para o formato de exibição `{}` , verá a seguinte saída:

   ```
   ( 1.1 1.2 )
   ( 2.1 2.2 )
   ```

   Talvez você queira consultar novamente o exemplo de exibição de impressão .

2. Adicione uma `transpose` função usando a `reverse` função como modelo, que aceita uma matriz como argumento e retorna uma matriz na qual dois elementos foram trocados. Por exemplo:

   ```
   println!("Matrix:\n{}", matrix);
   println!("Transpose:\n{}", transpose(matrix));
   ```

   Resultados na saída:

   ```
   Matrix:
   ( 1.1 1.2 )
   ( 2.1 2.2 )
   Transpose:
   ( 1.1 2.1 )
   ( 1.2 2.2 )
   ```

Matrizes e fatias

Um array é uma coleção de objetos do mesmo tipo `T`, armazenados em memória contígua. Os arrays são criados usando colchetes `[]`, e seu comprimento, conhecido em tempo de compilação, faz parte de sua assinatura de tipo `[T; length]`.

Fatias são semelhantes a matrizes, mas seu comprimento não é conhecido em tempo de compilação. Em vez disso, uma fatia é um objeto de duas palavras; a primeira palavra é um ponteiro para os dados, a segunda palavra é o comprimento da fatia. O tamanho da palavra é o mesmo que usize, determinado pela arquitetura do processador, por exemplo, 64 bits em um x86-64. Fatias podem ser usadas para pegar emprestado uma seção de uma matriz e ter a assinatura de tipo `&[T]`.

```rust
use std::mem;

// This function borrows a slice.
fn analyze_slice(slice: &[i32]) {
    println!("First element of the slice: {}", slice[0]);
    println!("The slice has {} elements", slice.len());
}

fn main() {
    // Fixed-size array (type signature is superfluous).
    let xs: [i32; 5] = [1, 2, 3, 4, 5];

    // All elements can be initialized to the same value.
    let ys: [i32; 500] = [0; 500];

    // Indexing starts at 0.
    println!("First element of the array: {}", xs[0]);
    println!("Second element of the array: {}", xs[1]);

    // `len` returns the count of elements in the array.
    println!("Number of elements in array: {}", xs.len());

    // Arrays are stack allocated.
    println!("Array occupies {} bytes", mem::size_of_val(&xs));

    // Arrays can be automatically borrowed as slices.
    println!("Borrow the whole array as a slice.");
    analyze_slice(&xs);

    // Slices can point to a section of an array.
    // They are of the form [starting_index..ending_index].
    // `starting_index` is the first position in the slice.
    // `ending_index` is one more than the last position in the slice.
    println!("Borrow a section of the array as a slice.");
    analyze_slice(&ys[1 .. 4]);

    // Example of empty slice `&[]`:
    let empty_array: [u32; 0] = [];
    assert_eq!(&empty_array, &[]);
    assert_eq!(&empty_array, &[][..]); // Same but more verbose

    // Arrays can be safely accessed using `.get`, which returns an
    // `Option`. This can be matched as shown below, or used with
    // `.expect()` if you would like the program to exit with a nice
    // message instead of happily continue.
    for i in 0..xs.len() + 1 { // Oops, one element too far!
        match xs.get(i) {
            Some(xval) => println!("{}: {}", i, xval),
            None => println!("Slow down! {} is too far!", i),
        }
    }

    // Out of bound indexing on array with constant value causes compile ti
    //println!("{}", xs[5]);
    // Out of bound indexing on slice causes runtime error.
    //println!("{}", xs[..][5]);
}
```