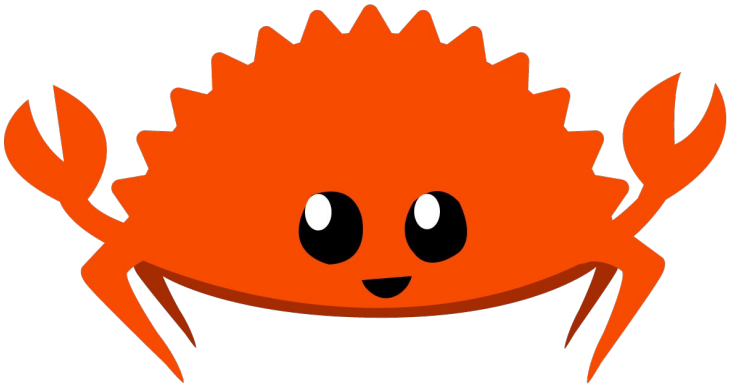
Nightly Night: Const Generics

By Jonathan Louie

Outline

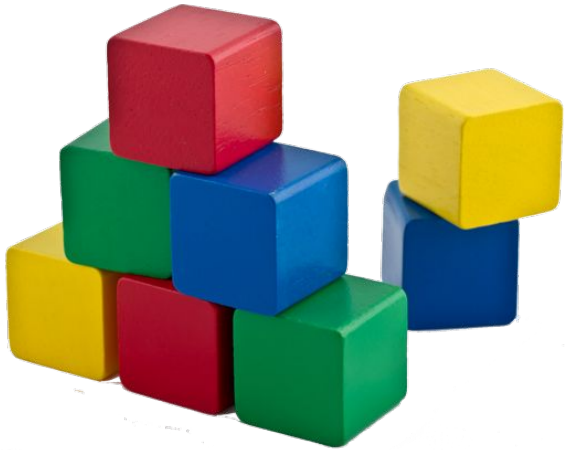
- Basics
- Const generics
- Const generics for type safety
- Current limitations
- What's landing in stable?



Basics

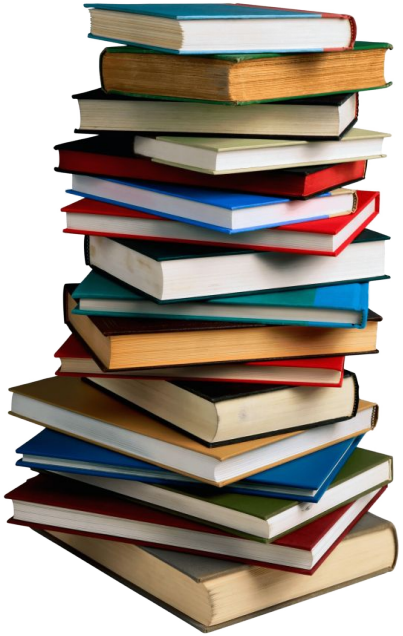
The Basics

- **Generics**
- **Arrays in Rust**
- **Constant evaluation in Rust (Constant Evaluation - The Rust Reference)**



Generics

```
fn head<A>(mut xs: Vec<A>) -> A {
    xs.pop().unwrap()
}

// code generated by head(vec![1])
fn head(mut xs: Vec<u32>) -> u32 {
    xs.pop().unwrap()
}

// code generated by head(vec![true])
fn head(mut xs: Vec<bool>) -> bool {
    xs.pop().unwrap()
}
```



Arrays

- **Parameterized by their length, which is a value**
  - **Types dependent on values are what people usually mean by "dependent types"**
- **Compiler catches index out of bounds errors if it knows the index at compile-time**
- **Example: https://play.rust-lang.org/?version=nightly&mode=debug&edition=2018&gist=a073dabf4303333a115bd8233a74cc61**

Constant Evaluation

- Some expressions can be evaluated entirely at compile-time
  - In "const context"s, only constant expressions are allowed
  - Constant expressions may not be evaluated at compile-time when outside a const context
- Array type length expressions are a "const context"
  - Thus, array lengths have to be known at compile-time

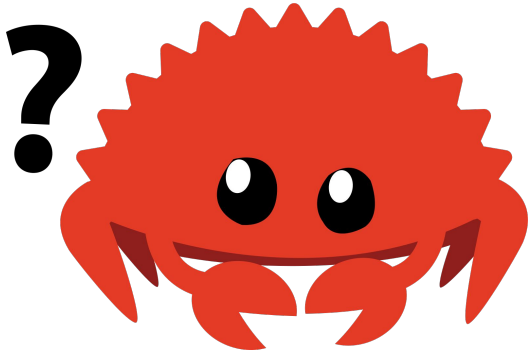
Example

```
fn main() {
    let xs: u32 = vec![2, 3, 4];

    // 3 is a const value so it can be used as the array length
    let arr: [u32; 3] = [1, 2, 3];

    // This won't compile because xs.len() is not a const value
    let arr: [u32; xs.len()] = unimplemented!();
}
```

Const Generics

What are const generics?

- Generics that are constant ***values***, not types
- Motivated by need to be generic over lengths of arrays
- Can be used for additional type safety
- Use this attribute to enable them: #![feature(const_generics)]



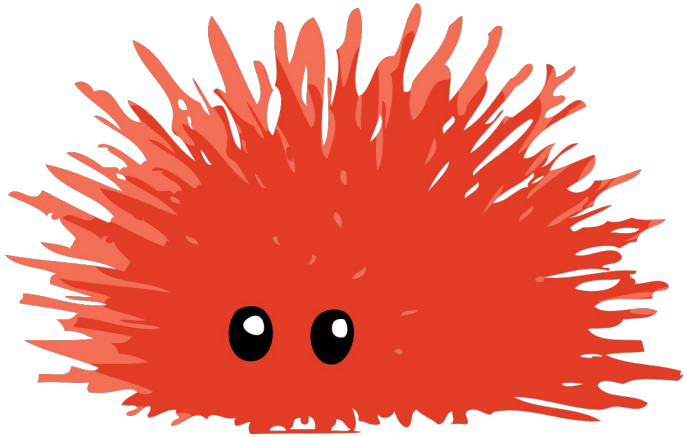
Why do we need const generics?

- **We can’t implement traits for all types of arrays**
  - **Example:** https://play.rust-lang.org/?version=nightly&mode=debug&edition=2018&gist=e755079bfb3a10a2f3e62f93dcbb0761
- **Working version of above example using const generics:** https://play.rust-lang.org/?version=nightly&mode=debug&edition=2018&gist=703df928a3e3875c3520103acf868ba2

Const Generics for Type Safety

Const Generics for Type Safety

- **Pre-conditions and post-conditions**
- **Example: (adapted from The Typestate Pattern in Rust): https://play.rust-lang.org/?version=nightly&mode=debug&edition=2018&gist=98885feef1ff2bda6a9653515dd68978**



Const generics vs. type-level programming

Const generics

- “Statically typed”
- Cleaner error messages
- Type-level numbers are concise
  - 5
- Limited to const expressions

Type-level programming

- "Dynamically typed"
- Messy error messages
- Type-level numbers are verbose
  - S<S<S<S<S<Z>>>>>
- Not limited to const expressions

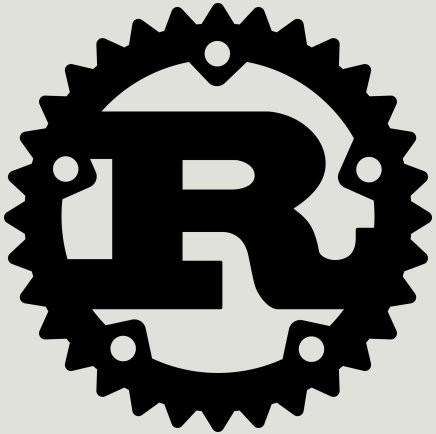
Const generics vs. dependent types

Const generics

- Types can only depend on constant values
- Only cover subset of dependent types

Dependent types

- Types can depend on runtime values
  - Values treated generically
- Can be used in theorem proving



Current limitations

Current Limitations

- **This is not allowed:**

```
struct Stack<const N: usize> {
    stack: [u64; {N / 8}],
}
```

- **You cannot currently (on nightly) use const generics in const expressions**
- **Fun fact: this was allowed before, but it was realized that this could cause an ICE in some cases so it was removed**
    - **I had to change my slides because of this...**

What’s landing in stable?

What will land in stable in 2020?

- **Status update from the RFC author: Shipping Const Generics in 2020**
  - **Example from post: https://play.rust-lang.org/?version=nightly&mode=debug&edition=2018&gist=39eb101fcf2b7458e052c117fbdb6edf**
- **A small subset of const generics**
  - **Only signed and unsigned integers, bools, chars**
  - **No complex expressions based on const generics**
- **Issue tracking remaining work for this subset: https://github.com/rust-lang/rust/issues/74878**

Questions?

Thanks for listening!