**Learning Rust: Const Generics**

[[](https://amacal.medium.com/?source=post_page-----8c29fc26fad4--------------------------------)](https://amacal.medium.com/?source=post_page-----8c29fc26fad4--------------------------------)

[Adrian Macal](https://amacal.medium.com/?source=post_page-----8c29fc26fad4--------------------------------)

·

Follow

3 min read

·

Jun 19

1



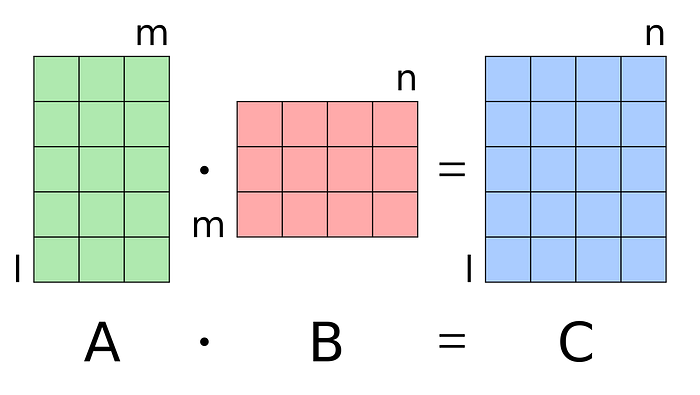
Don’t underestimate the power of studying others’ code; even the most familiar features can hold surprising learnings.

I accidentally stumbled upon an interesting piece of code, and it made me curious about what it is and how it works. I found this snippet in the [**arrayvec**](https://crates.io/crates/arrayvec) when I was trying to understand the main differences in this vector implementation. I discovered this piece of code, and const CAP: usize really grabbed my attention:

/// A vector with a fixed capacity.  
///  
/// The `ArrayVec` is a vector backed by a fixed size array. It keeps track of  
/// the number of initialized elements. The `ArrayVec<T, CAP>` is parameterized  
/// by `T` for the element type and `CAP` for the maximum capacity.  
///  
/// `CAP` is of type `usize` but is range limited to `u32::MAX`; attempting to create larger  
/// arrayvecs with larger capacity will panic.  
///  
/// The vector is a contiguous value (storing the elements inline) that you can store directly on  
/// the stack if needed.  
///  
/// It offers a simple API but also dereferences to a slice, so that the full slice API is  
/// available. The ArrayVec can be converted into a by value iterator.  
pub struct ArrayVec<T, const CAP: usize> {  
 // the `len` first elements of the array are initialized  
 xs: [MaybeUninit<T>; CAP],  
 len: LenUint,  
}

The answer was not hard to find. ChatGPT explained it was about const generics, a feature introduced in [Rust 1.5.1](https://blog.rust-lang.org/2021/03/25/Rust-1.51.0.html) (two years ago). This feature lets us use const values in generic definitions to create new concrete types during compilation time. In this case, it lets us define a fixed-size array within a struct without having to allocate it somewhere else on the heap and then exposing it with a Box.

This cool feature helps us write even more stable code. Just think about matrix multiplication. There’s a mathematical rule that says which kinds of matrices can be multiplied: the number of columns in the first matrix has to be equal to the number of rows in the second one.



<https://en.wikipedia.org/wiki/Matrix_multiplication>

The crate [mtrx](https://crates.io/crates/mtrx) uses this feature, but sadly, it doesn’t work right now. Look at this code. The compiler makes sure that matrices have the correct dimensions for multiplication. If you mess up with them, it won’t compile:

impl<const R: usize, const C: usize> Matrix<R, C> {  
 fn new(inner: [[i32; C]; R]) -> Self {  
 Matrix { inner }  
 }  
  
 fn dot\_product<const K: usize>(&self, row: usize, matrix: &Matrix<C, K>, col: usize) -> i32 {  
 let mut sum = self.inner[row][0] \* matrix.inner[0][col];  
  
 for i in 1..C {  
 sum += self.inner[row][i] \* matrix.inner[i][col]  
 }  
  
 sum  
 }  
  
 fn multiply<const K: usize>(&self, matrix: &Matrix<C, K>) -> Matrix<R, K> {  
 let mut inner = [[0; K]; R];  
  
 for r in 0..R {  
 for c in 0..K {  
 inner[r][c] = self.dot\_product(r, &matrix, c);  
 }  
 }  
  
 Matrix { inner }  
 }  
}  
  
fn main() {  
 let a = Matrix::new(  
 [[1, 2, 3],   
 [4, 5, 6]]  
 );  
   
 let b = Matrix::new(  
 [[7, 8],  
 [9, 10],   
 [11, 12]]  
 );  
   
 let result: Matrix<2, 2> = a.multiply(&b);  
 assert\_eq!(result.inner, [[58, 64], [139, 154]]);  
}

If you try to multiply matrices that can’t be multiplied, compiler generates an error. Amazing!

error[E0308]: mismatched types  
 --> src/main.rs:131:43  
 |  
131 | let result: Matrix<2, 2> = a.multiply(&b);  
 | -------- ^^ expected `3`, found `2`  
 | |  
 | arguments to this method are incorrect  
 |  
 = note: expected reference `&Matrix<3, 2>`  
 found reference `&Matrix<2, 2>`  
note: method defined here  
 --> src/main.rs:107:8  
 |  
107 | fn multiply<const K: usize>(&self, matrix: &Matrix<C, K>) -> Matrix<R, K> {  
 | ^^^^^^^^ ---------------------

[Rust](https://medium.com/tag/rust?source=post_page-----8c29fc26fad4---------------rust-----------------)