Static Assertions

Correctness and Stability via Cursed Code

by Nikolai Vazquez

Who am I?

Nikolai Vazquez, a.k.a. nvzqz

- 4th year Computer Science student at Boston University
- Open sourcerer since 2015
- Programming and natural languages enthusiast
- Cursed code connoisseur
- Graphic designer, pianist, and Oxford Comma evangelist
- Working on Swift and Rust playing nicely together

@NikolaiVazquez

What is “Static Assertions”?

- A Rust library hosted at github.com/nvzqz/static-assertions-rs
- Ensures at compile-time that:
  - Constant conditions are true
  - Types have the same size or alignment
  - All or any traits in a set are or are not implemented for a type
  - Traits support dynamic dispatch (object safety)
- All in user code; no compiler hacks

How You'll Walk Away From This

Pictured: Michael Gattozzi reading through one of this crate's macro implementations

@NikolaiVazquez

const_assert

@NikolaiVazquez

const_assert

```rust
const VALUE: i32
    = include!(concat!(env!("OUT_DIR"), "/value"));

const_assert!(VALUE > 42);
```

const_assert

```rust
const VALUE: i32
    = include!(concat!(env!("OUT_DIR"), "/value"));

const_assert!(VALUE > 42 && VALUE < 9000);
```

const_assert

```rust
const VALUE: i32
    = include!(concat!(env!("OUT_DIR"), "/value"));

const CONDITION: bool = VALUE > 42 && VALUE < 9000;

const_assert!(CONDITION);
```

const_assert

```
const_assert!(CONDITION);
```

```
error[E0080]: evaluation of constant value failed
  --> talk/const_assert.rs:4:1
   |
4  | const_assert!(CONDITION);
   | ^^^^^^^^^^^^^^^^^^^^^^^^^ attempt to subtract with overflow
```

const_assert

```
macro_rules! const_assert {
    ($x:expr) => {
        const _: [
            ();
            0 - !{
                const ASSERT: bool = $x;
                ASSERT
            } as usize
        ] = [];
    };
}
```

const_assert

```
macro_rules! const_assert {
    ($x:expr) => {
        const _: [
            ();
            0 - !{
                const ASSERT: bool = $x;
                ASSERT
            } as usize
        ] = [];
    };
}
```

@NikolaiVazquez

const_assert

```
macro_rules! const_assert {
    ($x:expr) => {
        const _: [
            ();
            0 - !{
                const ASSERT: bool = $x;
                ASSERT
            } as usize
        ] = [];
    };
}
```

@NikolaiVazquez

const_assert

```
macro_rules! const_assert {
    ($x:expr) => {
        const _: [
            ();
            0 - !{
                const ASSERT: bool = $x;
                ASSERT
            } as usize
        ] = [];
    };
}
```

@NikolaiVazquez

const_assert

```
const CONDITION: bool = // ...

const _: [(); 0 - !CONDITION as usize] = [];
```

const_assert

C++ Equivalent

```
#define STATIC_ASSERT(x) \
    typedef int __assert[(!!(x)) ? 1 : -1];
```

assert_eq_size

@NikolaiVazquez

assert_eq_size

```
assert_eq_size!(usize, *const u8);
```

assert_eq_size

```rust
assert_eq_size!(usize, *const u8, u32);
```

assert_eq_size

```rust
assert_eq_size!(usize, *const u8, u32);
```

```
error[E0512]: cannot transmute between types
of different sizes, or dependently-sized types
 --> tests/eq_size.rs:2:1
  |
2 | assert_eq_size!(usize, *const u8, u32);
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: source type: `usize` (64 bits)
  = note: target type: `u32` (32 bits)
```

assert__eq__size

```
macro_rules! assert_eq_size {
    ($x:ty, $($xs:ty),+) => {
        const _: fn() = || {
            use std::mem::transmute;

            $(let _ = transmute::<$x, $xs>;)+
        };
    }
```

@NikolaiVazquez

assert_eq_size

```
macro_rules! assert_eq_size {
    ($x:ty, $($xs:ty),+) => {
        const _: fn() = || {
            use std::mem::transmute;

            $(let _ = transmute::<$x, $xs>;)+
        };
    };
}
```

@NikolaiVazquez

assert__eq__size

```
macro_rules! assert_eq_size {
    ($x:ty, $($xs:ty),+) => {
        const _: fn() = || {
            use std::mem::transmute;

            $(let _ = transmute::<$x, $xs>;)+
        };
    }
}
```

@NikolaiVazquez

assert_eq_size

```
macro_rules! assert_eq_size {
    ($x:ty, $($xs:ty),+) => {
        const _: fn() = || {
            use std::mem::transmute;

            $(let _ = transmute::<$x, $xs>;)+
        };
    };
}
```

@NikolaiVazquez

assert_eq_align

@NikolaiVazquez

assert_eq_align

```rust
assert_eq_align!(usize, *const u8, u32);
```

assert_eq_align

```
assert_eq_align!(usize, *const u8, u32);
```

```
error[E0308]: mismatched types
  --> tests/eq_size.rs:2:1
   |
2  | assert_eq_align!(usize, *const u8, u32);
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   | expected an array with a fixed size of 8 elements,
   | found one with 4 elements
   |
   = note: expected type `[(); 8]`
              found type `[(); 4]`
```

assert__eq__align

```
macro_rules! assert_eq_align {
    ($x:ty, $($xs:ty),+) => {
        const _: fn() = || {
            use $crate::_core::mem::align_of;

            $(let _: [(); align_of::<$x>()]
                    = [(); align_of::<$xs>()];)+
        };
    };
}
```

@NikolaiVazquez

assert__eq__align

```
macro_rules! assert_eq_align {
    ($x:ty, $($xs:ty),+) => {
        const _: fn() = || {
            use $crate::_core::mem::align_of;

            $(let _: [(); align_of::<$x>()]
                  = [(); align_of::<$xs>()];)+
        };
    };
}
```

@NikolaiVazquez

assert__eq__align

```
macro_rules! assert_eq_align {
    ($x:ty, $($xs:ty),+) => {
        const _: fn() = || {
            use $crate::_core::mem::align_of;

            $(let _: [(); align_of::<$x>()]
                    = [(); align_of::<$xs>()];)+
        };
    };
}
```

@NikolaiVazquez

assert_eq_align

```
macro_rules! assert_eq_align {
    ($x:ty, $($xs:ty),+) => {
        const _: fn() = || {
            use $crate::_core::mem::align_of;

            $(let _: [(); align_of::<$x>()]
                  = [(); align_of::<$xs>()];)+
        };
    };
}
```

@NikolaiVazquez

assert_not_impl_all

@NikolaiVazquez

assert_not_impl_all

```rust
struct Foo;

trait Bar {}
trait Baz {}

impl Bar for Foo {}
impl Baz for Foo {}

assert_not_impl_all!(Foo: Bar, Baz);
```

assert_not_impl_all

```
assert_not_impl_all!(Foo: Bar, Baz);
```

```
error[E0282]: type annotations needed for
`fn() {<Foo as _::{{closure}}#0::AmbiguousIfImpl<_>>::some_item}`
  --> talk/not_impl_all.rs:10:1
   |
10 | assert_not_impl_any!(Foo: Bar, Baz);
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   | |
   | cannot infer type
   | consider giving this pattern the explicit type
   | `fn() {<Foo as _::{{closure}}#0::AmbiguousIfImpl<_>>::some_item}`,
   | with the type parameters specified
```

assert__not__impl__all

```
macro_rules! assert_not_impl_all {
    ($x:ty: $($t:path),+) => {
        const _: fn() = || {
            struct Invalid;

            trait AmbiguousIfImpl<A> {
                fn some_item() {}
            }

            impl<T: ?Sized>
                AmbiguousIfImpl<()> for T {}

            impl<T: ?Sized $(+ $t)+>
                AmbiguousIfImpl<Invalid> for T {}

            let _ = <$x as AmbiguousIfImpl<_>>::some_item;
        };
    };
}
```

@NikolaiVazquez

assert__not__impl__all

```
macro_rules! assert_not_impl_all {
    ($x:ty: $($t:path),+) => {
        const _: fn() = || {
            struct Invalid;

            trait AmbiguousIfImpl<A> {
                fn some_item() {}
            }

            impl<T: ?Sized>
                AmbiguousIfImpl<()> for T {}

            impl<T: ?Sized $(+ $t)+>
                AmbiguousIfImpl<Invalid> for T {}

            let _ = <$x as AmbiguousIfImpl<_>>::some_item;
        };
    };
}
```

@NikolaiVazquez

assert_not_impl_all

```
macro_rules! assert_not_impl_all {
    ($x:ty: $($t:path),+) => {
        const _: fn() = || {
            struct Invalid;

            trait AmbiguousIfImpl<A> {
                fn some_item() {}
            }

            impl<T: ?Sized>
                AmbiguousIfImpl<()> for T {}

            impl<T: ?Sized $(+ $t)+>
                AmbiguousIfImpl<Invalid> for T {}

            let _ = <$x as AmbiguousIfImpl<_>>::some_item;
        };
    };
}
```

@NikolaiVazquez

assert__not__impl__all

```
macro_rules! assert_not_impl_all {
    ($x:ty: $($t:path),+) => {
        const _: fn() = || {
            struct Invalid;

            trait AmbiguousIfImpl<A> {
                fn some_item() {}
            }

            impl<T: ?Sized>
                AmbiguousIfImpl<()> for T {}

            impl<T: ?Sized $(+ $t)+>
                AmbiguousIfImpl<Invalid> for T {}

            let _ = <$x as AmbiguousIfImpl<_>>::some_item;
        };
    };
}
```

@NikolaiVazquez

Fin

@NikolaiVazquez