Rust Undefined Behavior

2025-10-01

Florian “Florob” Zeitz

Outline

1 Demystifying Undefined Behavior 3
2 Uninitialized Data 23
3 Arithmetic 29
4 Provenance and Aliasing 39
5 Writing Unsafe Rust 62

1 Demystifying Undefined Behavior

Quiz: After main

How does this snippet behave? (clang 20, -O1)

```c
int main(void) {
    for (int i = 0; i >= 0; i++)
        ;
}

void after(void) {
    puts("Hello World");
}
```

A terminates

B runs forever

C prints, terminates

D prints, runs forever

Quiz: After main

How does this snippet behave? (clang 20, -O1)

```c
int main(void) {
    for (int i = 0; i >= 0; i++)
        ;
}

void after(void) {
    puts("Hello World");
}
```

A terminates

B runs forever

C prints, terminates

D prints, runs forever

Definition: C23 (ISO/IEC 9899:2024)

undefined behavior

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this document imposes no requirements

Note 1 to entry: Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). […]

— C23

- no requirements imposed
- upon use of:
  - nonportable/erroneous program constructs
  - erroneous data

- possible undefined behavior
  - ignore the situation completely
  - behave in a documented manner
  - abort translation or execution

Definition: Rust Unsafe Code Guidelines

> *Undefined Behavior* is a concept of the contract between the Rust programmer and the compiler: The programmer promises that the code exhibits no undefined behavior. In return, the compiler promises to compile the code in a way that the final program does on the real hardware what the source program does according to the Rust Abstract Machine. If it turns out the program does have undefined behavior, the contract is void, and the program produced by the compiler is essentially garbage (in particular, it is not bound by any specification; the program does not even have to be well-formed executable code).
>
> In Rust, the Nomicon and the Reference both have a list of behavior that the language considers undefined. Rust promises that safe code cannot cause Undefined Behavior—the compiler and authors of unsafe code takes the burden of this contract on themselves. For unsafe code, however, the burden is still on the programmer.
>
> — Rust Unsafe Code Guidelines Glossary

- contract between programmer and compiler
  - programmer: no undefined behavior
  - compiler: final program behaves as source describes
- safe code cannot have undefined behavior
- unsafe code places burden on the programmer

Perception: What They Say

> Learning that the compiler recognizes UB but instead of point it out concludes that it should be taken out of the program for speed reasons was insane to me
>
> — A redditor

> Any instance of undefined behavior should result in a Warning, if not an Error.
>
> — A HN User

> […] compiler writers […] think that if the programmer even approaches anything undefined, they can do what ever, completely disregarding, if it makes logical sense, if it is predictable behavior or is in anyway useful to software development.
>
> — A Tech Blogger

Reality

- all of this is false
- rarely does a compiler actually detect UB
- instead compilers assume UB does not occur
- some instances of UB are not detectable
- detected UB usually *does* yield a warning

Assembly Output


```
1  main:
2      # Nothing here
3
4  after:
5      leaq .L.str(%rip), %rdi
6      jmp puts@PLT # TAILCALL
7
8  .L.str:
9      .asciz "Hello World"
10     .size .L.str, 12
```

Optimization: Initial IR (simplified)


```
@.str = [12 x i8] c"Hello World\00", align 1

; Function Attrs: nounwind sspstrong uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 0, ptr %2, align 4, !tbaa !5
  br label %3

; loop head
3:                                ; preds = %8, %0
  %4 = load i32, ptr %2, align 4, !tbaa !5
  %5 = icmp sge i32 %4, 0
  br i1 %5, label %8, label %11

; loop body
8:                                ; preds = %3
  %9 = load i32, ptr %2, align 4, !tbaa !5
  %10 = add nsw i32 %9, 1
  store i32 %10, ptr %2, align 4, !tbaa !5
  br label %3, !llvm.loop !9

11:                               ; preds = %3
  %12 = load i32, ptr %1, align 4
  ret i32 %12
}

; Function Attrs: nounwind sspstrong uwtable
define dso_local void @after() #0 {
  %1 = call i32 @puts(ptr noundef @.str)
  ret void
}

!5 = !{!6, !6, i64 0}
!6 = !{!"int", !7, i64 0}
!7 = !{!"omnipotent char", !8, i64 0}
!8 = !{!"Simple C/C++ TBAA"}
!9 = distinct !{!9, !10, !11}
!10 = !{!"llvm.loop.mustprogress"}
!11 = !{!"llvm.loop.unroll.disable"}
```

Optimization: After SROA


```
1 ; Function Attrs: nounwind sspstrong uwtable
2 define dso_local i32 @main() #0 {
3   br label %1
4
5 ; loop head
6 1:                                ; preds = %5, %0
7   %2 = phi i32 [ 0, %0 ], [ %6, %5 ]
8   %3 = icmp sge i32 %2, 0
9   br i1 %3, label %5, label %4
10
11 4:                               ; preds = %1
12  ret i32 0
13
14 ; loop body
15 5:                               ; preds = %1
16  %6 = add nsw i32 %2, 1
17  br label %1, !llvm.loop !5
18 }
```

SROA

Scalar Replacement of Aggregates

Optimization: After EarlyCSE

```
; Function Attrs: nounwind sspstrong uwtable
define dso_local i32 @main() #0 {
  br label %1

; loop head
1:                                ; preds = %4, %0
  %2 = phi i32 [ 0, %0 ], [ %5, %4 ]
  br i1 true, label %4, label %3

3:                                ; preds = %1
  ret i32 0

; loop body
4:                                ; preds = %1
  %5 = add nsw i32 %2, 1
  br label %1, !llvm.loop !5
}
```

EarlyCSE

Early Common Sub-expression Elimination

Optimization: After IPSCCP


```
; Function Attrs: nounwind sspstrong uwtable
define dso_local i32 @main() #0 {
  br label %1

1:                                ; preds = %3, %0
  %2 = phi i32 [ 0, %0 ], [ %4, %3 ]
  br label %3

3:                                ; preds = %1
  %4 = add nsw i32 %2, 1
  br label %1, !llvm.loop !5
}

!5 = distinct !{!5, !6, !7}
!6 = !{!"llvm.loop.mustprogress"}
!7 = !{!"llvm.loop.unroll.disable"}
```

IPSCCP

Interprocedural Sparse Conditional Constant Propagation

Optimization: After InstCombine and SimplifyCFG

```
; Function Attrs: nounwind sspstrong uwtable
define dso_local i32 @main() local_unnamed_addr #0 {
  br label %1

1:                                ; preds = %1, %0
  br label %1, !llvm.loop !5
}

!5 = distinct !{!5, !6, !7}
!6 = !{!"llvm.loop.mustprogress"}
!7 = !{!"llvm.loop.unroll.disable"}
```

SimplifyCFG

Simplify Control Flow Graph

Optimization: After LoopDeletion


```
1 ; Function Attrs: nounwind sspstrong uwtable
2 define dso_local i32 @main() local_unnamed_addr #0 {
3   unreachable
4 }
```

LoopDeletion

deletes loop if:

- Produced values are loop invariant
- All exits produce the same value
- No instructions have side-effects
- The loop must progress (stop here if true)
- The loop is definitely not infinite

What About Termination?

- Shouldn't we always check for termination?
- Loops may be finite, but not provably so
- Trade-off:
  - Always remove dead loops, more UB
  - Keep some dead loops, less UB
  - This is a general pattern

Can this happen in Rust?

- Can we get the same fallthrough in Rust?
- No C-style loops
- Infinite loops are well-defined (we even have `loop`)
- Maybe we can work with `unreachable`

Reproduction: unreachable!()

`unreachable!()` panics, so doesn't work

```rust
fn main() {
    unreachable!();
}

pub fn after() {
    println!("Hello World");
}
```

```
main:
    pushq %rax
    leaq .L1(%rip), %rdi
    leaq .L3(%rip), %rdx
    movl $40, %esi
    callq *panic@GOTPCREL(%rip)

.L1:
    .ascii "internal error: ..."
    .size .L1, 40
...
```

Reproduction: unreachable_unchecked()

`unreachable_unchecked()` emits `ud2` ⟹ always generates an exception

```rust
fn main() {
    unsafe {
        unreachable_unchecked();
    }
}

pub fn after() {
    println!("Hello World")
}
```

```
main:
    ud2
```

Reproduction: noreturn

`noreturn` also emits `ud2` ⟹ always generates an exception

```rust
use std::arch::asm;

fn main() {
    unsafe {
        asm!("", options(noreturn));
    }
}

pub fn after() {
    println!("Hello World")
}
```

```
main:
    ud2
```

Reproduction: Conclusion

- Rust configures LLVM to emit `ud2` for `unreachable`
- This avoids unexpected results, even though this *is* undefined behavior
- In binary crates Rust removes unused functions, even if they are `pub`

2 Uninitialized Data

Quiz: a or b

What does this snippet usually print? (optimized)

```c
int a, b;

if (a) {
    a = 3;
} else {
    b = 4;
}

printf("%i\n", a + b);
```

A 3

B 4

C 5

D 7

Quiz: a or b

What does this snippet usually print? (optimized)

```c
int a, b;

if (a) {
    a = 3;
} else {
    b = 4;
}

printf("%i\n", a + b);
```

A 3

B 4

C 5

D 7

Uninitialized Data in Safe Rust

- Impossible in safe Rust
- Compiler forbids usage of possibly uninitialized variables

Uninitialized Data in Unsafe Rust


```rust
fn main() {
    let mut a = MaybeUninit::<i32>::uninit();
    let mut b = MaybeUninit::<i32>::uninit();

    if std::env::var("A").is_ok() {
        a.write(3);
    } else {
        b.write(4);
    }

    let res = unsafe { a.assume_init() + b.assume_init() };
    println!("{res}");
}
```

- Same undefined behavior as the C version

Properly Using MaybeUninit


```rust
let data = {
    let mut data: [MaybeUninit<Vec<u32>>; 1000]=
        [const { MaybeUninit::uninit() }; 1000];

    // A panic in this loop would leak memory,
    // dropping `MaybeUninit` does not drop its contents
    for elem in &mut data[..] {
        elem.write(vec![42]);
    }

    unsafe { mem::transmute::<_, [Vec<u32>; 1000]>(data) }
};
```

- Uninitialized memory in Rust must be contained in `MaybeUninit`
- Calling `assume_init()` when the content is not fully initialized causes immediate UB
- Cf. `MaybeUninit`'s documentation

3 Arithmetic

Quiz: Addition

What does this snippet usually print when size is `INT_MAX`? (optimized with -O3)

```
1 int size = ...;
2 if (size > size+1) {
3   puts("Aborted")
4   abort();
5 }
6 puts("Fetching memory");
7 malloc(size+1);
```

A “Fetching memory”

B “Aborted”

C size

D Nothing

Quiz: Addition

What does this snippet usually print when size is `INT_MAX`? (optimized with -O3)

```
1 int size = ...;
2 if (size > size+1) {
3   puts("Aborted")
4   abort();
5 }
6 puts("Fetching memory");
7 malloc(size+1);
```

A “Fetching memory”

B “Aborted”

C size

D Nothing

Signed Integer Overflow (C)

- unsigned integer overflow is well-defined: `UINT_MAX + 1 == 0`
- signed integer overflow is not: `INT_MAX + 1 == /* undef */`
- rumours aside `INT_MAX + 1` is **not** `INT_MIN`
- Check equality against `INT_MAX`

Signed Integer Overflow (C)


```c
int size = ...;
if (size > size+1) {
  puts("Aborted")
  abort();
}
puts("Fetching memory");
malloc(size+1);
```

- Only defined behavior is considered
- `size > size + 1` is always false
- Optimization removes the branch

Integer Overflow (Rust)

- same behaviour for all integer types, signed and unsigned
- debug: panic on overflow
- release: wrap around on overflow
- individual methods for specific requirements:
  - `checked_add()`
  - `saturating_add()`
  - `wrapping_add()`
  - `overflowing_add()`

Quiz: Shift

What does this snippet usually print? (unoptimized, x86)

```
1 uint32_t shifty = 1;
2 shifty = shifty << 32;
3 printf("%"PRIu32"\n", shifty);
```

A 0

B 1

C 4

D 32

Quiz: Shift

What does this snippet usually print? (unoptimized, x86)

```
1 uint32_t shifty = 1;
2 shifty = shifty << 32;
3 printf("%"PRIu32"\n", shifty);
```

A 0

B 1

C 4

D 32

Oversized Shift Amounts (C)

> If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.
>
> — C23

- Set the variable to zero instead
- Trivial to check when the type-width is known

Oversized Shift Amounts (Rust)

- debug: panic on oversized shift amount
- release: mask right operand to bit width
- individual methods for specific requirements:
  - `checked_shl()`
  - `wrapping_shl()`
  - `overflowing_shl()`

4 Provenance and Aliasing

Quiz: (Wh)y?

What does this print? (Rust 1.90, optimized)

```rust
let mut x = 42u8;
let mut y = 42u8;

let diff = (&raw const y)
    .offset_from(&raw const x);
let px = (&raw mut x)
    .wrapping_offset(diff);
let py = &raw mut y;

if ptr::eq(px, py) {
    *px = 10;
    println!(
        "{}, {}, {}, {}",
        x, y, *px, *py
    );
}
```

A 42, 10, 10, 10

B Nothing

C 42, 42, 10, 42

D 10, 42, 42, 42

Quiz: (Wh)y?

What does this print? (Rust 1.90, optimized)

```rust
let mut x = 42u8;
let mut y = 42u8;

let diff = (&raw const y)
    .offset_from(&raw const x);
let px = (&raw mut x)
    .wrapping_offset(diff);
let py = &raw mut y;

if ptr::eq(px, py) {
    *px = 10;
    println!(
        "{}, {}, {}, {}",
        x, y, *px, *py
    );
}
```

A 42, 10, 10, 10

B Nothing

C 42, 42, 10, 42

D 10, 42, 42, 42

Provenance

- pointers in Rust are more than addresses
- in the abstract machine pointers consist of
  - an address
  - provenance
- provenance represents the memory a pointer is allowed to access
- provenance is inherited by derived pointers
- integers do not have provenance $\Longrightarrow$ bit hacks cause loss of provenance
- as of Rust 1.84 there are two ways to deal with this
  - strict provenance
  - exposed provenance
- Rust Doc

Strict Provenance

- can be used if you can keep a pointer with correct provenance around
- no integer-to-pointer casts
- `fn with_addr(self, addr: usize) -> Self` pointer with given address and provenance of `self`
- `fn addr(self) -> usize` get address portion of pointer
- `fn map_addr(self, f: impl FnOnce(usize) -> usize) -> Self` let a closure modify `self`'s address keeping provenance

Example: Tagged Pointer


```rust
#[derive(Clone, Copy)]
pub struct EnumPointer<'a> {
    ptr: *const (),
    _phantom: PhantomData<&'a ()>,
}

impl<'a> EnumPointer<'a> {
    const IS_U32: usize = 0b01;
    const IS_F32: usize = 0b10;
    const PTR_MASK: usize = !0b11;

    const _HAS_MIN_ALIGN: () = {
        assert!(mem::align_of::<u32>() >= 4);
        assert!(mem::align_of::<f32>() >= 4);
    };
}
```

Example: Tagged Pointer (Constructors)


```
impl EnumPointer<'_> {
    pub fn from_u32(x: &u32) -> EnumPointer<'_> {
        EnumPointer {
            ptr: ptr::from_ref(x).map_addr(|a| a | Self::IS_U32).cast(),
            _phantom: PhantomData,
        }
    }

    pub fn from_f32(x: &f32) -> EnumPointer<'_> {
        EnumPointer {
            ptr: ptr::from_ref(x).map_addr(|a| a | Self::IS_F32).cast(),
            _phantom: PhantomData,
        }
    }
}
```

Example: Tagged Pointer (Accessors)


```rust
impl<'a> EnumPointer<'a> {
    pub fn as_u32(self) -> Option<&'a u32> {
        (self.ptr.addr() & Self::IS_U32 != 0)
            .then(|| unsafe { &*self.ptr.map_addr(|a| a & Self::PTR_MASK).cast() })
    }

    pub fn as_f32(self) -> Option<&'a f32> {
        (self.ptr.addr() & Self::IS_F32 != 0)
            .then(|| unsafe { &*self.ptr.map_addr(|a| a & Self::PTR_MASK).cast() })
    }
}
```

Exposed Provenance

- allows integer to pointer cast
- unsure if this can be formalized unambiguously
- casting a pointer to an integer exposes its provenance
- casting an integer to a pointer can obtain any previously exposed provenance
- things like memory mapped IO are always considered exposed
- `fn expose_provenance(self) -> usize` equivalent to `self as usize`
- `fn with_exposed_provenance<T>(addr: usize) -> *const T` equivalent to `addr as *const T`
- `fn with_exposed_provenance_mut<T>(addr: usize) -> *mut T` equivalent to `addr as *mut T`

Quiz: Aliasing

What does this snippet usually print? (optimized, clang or gcc)

```
1  void f(int *i, float *f) {
2    *i = 42;
3    *f = 16.0;
4    printf("%x\n", *i);
5  }
6
```

```
7  int main(void) {
8    int var;
9    f(&var,  &var);
10   return 0;
11 }
```

A “2a”

B “10”

C “41800000”

D “0”

Quiz: Aliasing

What does this snippet usually print? (optimized, clang or gcc)

```c
void f(int *i, float *f) {
  *i = 42;
  *f = 16.0;
  printf("%x\n", *i);
}
```

```c
int main(void) {
  int var;
  f(&var,  &var);
  return 0;
}
```

A “2a”

B “10”

C “41800000”

D “0”

Strict Aliasing Rule

- C allows aliasing
- after `int *pa = &a`, `*pa` aliases `a`
- not all expression may be used to access an object
- expression and object type must match
- this restriction is commonly called the **strict aliasing rule**
- with `a` declared as a `float`, `*pa` may be neither read nor written

Strict Aliasing Exceptions

- different signedness
- different qualifiers
- struct, array or union type with a member of one of the aforementioned types
- character type

Why Strict Aliasing

- makes it harder to create trap/niche values (e.g. `bool` with value 5 in memory)
- avoids unaligned accesses
- restricts aliasing (potential for optimizations)

Aliasing and Safe Rust

- type punning is not possible in safe Rust
- Rust guarantees values cannot be mutated in the presence of aliasing
- aliasing in safe Rust is much more restricted than in C
- allows for more optimizations than strict aliasing

Quiz: Alalialiasing

How many possible results does this function have?

```
1 int f(signed int *i1, unsigned int *i2, float *f, char *c) {
2   *i1 = 42;
3   *i2 = 43;
4   *f = 13.;
5   *c = 1;
6   return *i1 + *i2 + *f + *c;
7 }
```

A 1

B 22

C 531

D 2^{17}

Quiz: Alalialiasing

How many possible results does this function have?

```
1 int f(signed int *i1, unsigned int *i2, float *f, char *c) {
2   *i1 = 42;
3   *i2 = 43;
4   *f = 13.;
5   *c = 1;
6   return *i1 + *i2 + *f + *c;
7 }
```

A 1

B 22

C 531

D 2^{17}

Legal aliasing

- None (+1)
- `c` and any byte of `i1` (+4)
- `c` and any byte of `i2` (+4)
- `c` and any byte of `f` (+4)
- `i1` and `i2` (+1)
- `i1` and `i2` and `c` with any byte of them (+4)
- `i1` and `i2`, and `c` with any byte of `f` (+4)
- 22 possibilities

Generated Assembly (C)


```c
int f(signed int *i1,
      unsigned int *i2,
      float *f,
      char *c)
{
    *i1 = 42;
    *i2 = 43;
    *f = 13.;
    *c = 1;
    return *i1 + *i2 + *f + *c;
}
```

```asm
.LCPI0_0:
   .long 0x3f800000 # float 1
f:
   movl $42, (%rdi)
   movl $43, (%rsi)
   movl $1095761920, (%rdx)
   movb $1, (%rcx)
   movl (%rsi), %eax
   addl (%rdi), %eax
   cvtsi2ss %rax, %xmm0
   addss (%rdx), %xmm0
   addss .LCPI0_0(%rip), %xmm0
   cvttss2si %xmm0, %eax
   retq
```

Generated Assembly (Rust)


```rust
fn f(
    i1: &mut i32, i2: &mut u32,
    f: &mut f32, c: &mut i8,
) -> i32 {
    *i1 = 42;
    *i2 = 43;
    *f = 13.;
    *c = 1;
    (
        *i1 as f32 + *i2 as f32
            + *f + *c as f32
    ) as i32
}
```

```asm
f:
    movl    $42, (%rdi)
    movl    $43, (%rsi)
    movl    $1095761920, (%rdx)
    movb    $1, (%rcx)
    movl    $99, %eax
    retq
```

Aliasing and Unsafe Rust

- arbitrary aliasing is possible in unsafe Rust
- we don't know what exactly is allowed (yet)
- optimizations in the presence of unsafe code have to be conservative
- ongoing work to specify an aliasing model

Stacked Borrows

- first(?) potential aliasing model for Rust
- lots of UB
- allows for many optimizations
- some widely used crates have UB under this model, e.g. tokio

Tree Borrows

- potential aliasing model for Rust by Neven Villani et al.
- less UB, allows situations the borrow checker forbids
- still allows for most optimizations, and even additional ones
- Paper
- PLDI'25 presentation
- Example Oriented Description

5 Writing Unsafe Rust

Writing Unsafe Rust

- writing unsafe Rust can be difficult
- have to watch out for UB
- we don't even know what exactly is or isn't UB in unsafe Rust
- Resources and Tools:
  - The Rustonomicon
  - Language Reference
  - Unsafe Code Guidelines (largely abandoned)
  - Miri

The Good News

- everything that is safe in safe Rust is still safe in an `unsafe` block
- `unsafe` blocks allow only a few additional operations:
  - dereference raw pointers
  - call `unsafe` functions
  - implement `unsafe` traits
  - access or modify mutable statics
  - access `union` fields

Miri

- interpreter that can detect undefined behavior
- detects:
  - out-of-bounds memory accesses and use-after-free
  - invalid use of uninitialized data
  - violation of intrinsic preconditions (e.g. `unreachable_unchecked()` being reached)
  - insufficiently aligned memory accesses and references
  - violation of some basic type invariants (e.g. `bool` that is not 0 or 1, invalid `enum` discriminant)
  - data races and some weak memory effects (e.g old value returned by atomic read)
  - **experimental**: violations of aliasing rules (according to Stacked or Tree Borrows)

Questions?



https://babelmonkeys.de/~florob/talks/RC-2025-10-01-unsafe-undefined.pdf

Thank you for your attention. Any questions?