Rust Undefined Behavior

Florian “Florob” Zeitz

2023-06-07

Definition

undefined behavior — C17

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International *Standard 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).

Definition

Undefined Behavior — 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.

Perception

A Redditor

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 HN User

Any instance of undefined behavior should result in a Warning, if not an Error.

A Tech Blogger

[...] 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.

Perception

- 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 often *does* yield a warning

What does this snippet behave?
(clang 15, -O1)

```
1  int main(void) {
2      for(int i = 0; i >= 0; i++)
3          ;
4  }
5
6  void after(void) {
7      puts("Hello World");
8  }
```

terminates

runs forever

prints, terminates

prints, runs forever

What does this snippet behave?
(clang 15, -O1)

```
1  int main(void) {
2      for(int i = 0; i >= 0; i++)
3          ;
4  }
5
6  void after(void) {
7      puts("Hello World");
8  }
```

terminates

runs forever

prints, terminates

prints, runs forever

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

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

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 on main

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

Optimization

After EarlyCSE on main

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

Optimization

After IPSCCP on module

```
1   ; Function Attrs: nounwind sspstrong uwtable
2   define dso_local i32 @main() #0 {
3     br label %1
4
5   1:                                ; preds = %3, %0
6     %2 = phi i32 [ 0, %0 ], [ %4, %3 ]
7     br label %3
8
9   3:                                ; preds = %1
10    %4 = add nsw i32 %2, 1
11    br label %1, !llvm.loop !5
12  }
13
14  !5 = distinct !{!5, !6, !7}
15  !6 = !{!"llvm.loop.mustprogress"}
16  !7 = !{!"llvm.loop.unroll.disable"}
```

Optimization

After InstCombine and SimplifyCFG on main

```
1   ; Function Attrs: nounwind sspstrong uwtable
2   define dso_local i32 @main() local_unnamed_addr #0 {
3     br label %1
4
5   1:                                ; preds = %1, %0
6     br label %1, !llvm.loop !5
7   }
8
9   !5 = distinct !{!5, !6, !7}
10  !6 = !{!"llvm.loop.mustprogress"}
11  !7 = !{!"llvm.loop.unroll.disable"}
```

Optimization

After LoopDeletion

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

Checks:

- 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

Why do this?

- 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

Reproducing the Result 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`

Reproducing the Result in Rust

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

```
1  fn main() {
2      unreachable!();
3  }
4
5  pub fn after() {
6      println!("Hello World");
7  }
```

```
1  main:
2    pushq %rax
3    leaq  .L2(%rip), %rdi
4    leaq  .L3(%rip), %rdx
5    movl  $40, %esi
6    callq *panic@GOTPCREL(%rip)
7    ud2
8
9  .L2:
10   .ascii "internal error:
     ↪  entered unreachable
     ↪  code"
11   ...
```

Reproducing the Result in Rust

`unreachable_unchecked()` emits `ud2`. Always generates an exception.

```
1  fn main() {
2      unsafe {
3          unreachable_unchecked();
4      }
5  }
6
7  pub fn after()  {
8      println!("Hello World");
9  }
```

```
1  main:
2      ud2
```

Reproducing the Result in Rust

`noreturn` also emits `ud2`.

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

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

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

```
main:
  pushq %rax
  ud2
```

Reproducing the Result in Rust

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

What does this snippet usually print? (optimized)

```
1 int a, b;
2 if (a) {
3   a = 3;
4 } else {
5   b = 4;
6 }
7 printf("%i\n", a + b);
```

3

4

5

7

What does this snippet usually print? (optimized)

```
1 int a, b;
2 if (a) {
3   a = 3;
4 } else {
5   b = 4;
6 }
7 printf("%i\n", a + b);
```

- 3
- 4
- 5
- 7

Uninitialized in safe Rust

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

Uninitialized 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}");
}
```

Uninitialized in unsafe Rust

- Uninitialized memory in Rust must be contained in `MaybeUninit`
- Calling `assume_init()` when the content is not fully initialized causes immediate UB
- The deprecated `std::mem::uninitialized()` caused the same UB
- Cf. `MaybeUninit`'s documentation

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);
```

- "Fetching memory"
- "Aborted"
- size
- Nothing

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);
```

- "Fetching memory"
- "Aborted"
- size
- 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`

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

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

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

0 | 1

4 | 32

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

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

0	1
4	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.*

- set variables to zero instead
- easily checked when 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()`

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

2a

10

41800000

0

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

- 2a
- 10
- 41800000
- 0

Strict Aliasing Rule

- C allows aliasing
- `int *pa = &a`, `*pa` aliases `a`
- not all expressions 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

Exceptions

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

Why have this rule?

- makes it harder to create trap/niche values
- avoids unaligned writes
- 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

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  }
```

1

19

531

2^{17}

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  }
```

1	19
531	2^{17}

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;
}
```

```
.LCPI0_0:
  .long 1065353216 # float 1
f:
  movl $42, (%rdi)
  movl $43, (%rsi)
  movl $1095761920, (%rdx)
  movb $1, (%rcx)
  movl (%rsi), %eax
  addl (%rdi), %eax
  cvtsi2ssq %rax, %xmm0
  addss (%rdx), %xmm0
  addss .LCPI0_0(%rip), %xmm0
  cvttss2si %xmm0, %eax
  retq
```

Generated Assembly (`&mut T`)


```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

- recent potential aliasing model for Rust
- less UB, allows situations the borrow checker forbids
- still allows for most optimizations, and even additional ones
- Ralf Jung's Blog Post
- Neven Villani's Description
- Recording of Neven Villani's Talk

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/Tools:
  - Unsafe Code Guidelines
  - The Rustonomicon
  - Language Reference
  - Miri

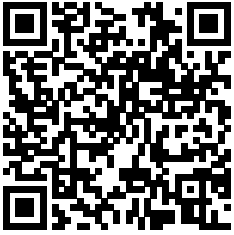
The Good News

- 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
  - mutate 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)
  - *experimental*: Violations of aliasing rules (according to Stacked or Tree Borrows)
  - *experimental*: Data races

Thank you for your attention. Any questions?



https://babelmonkeys.de/~florob/talks/RC-2023-06-07-unsafe-undefined.pdf