Undefined Behavior

Matt Torrence

Pop Quiz

```c
#include <stdio.h>

int main() {
    printf("%d\n", 1<<32);
    return 0;
}
```

If you were following the C standard to the “t”, what should this code print out:

Pop Quiz

```c
#include <stdio.h>

int main() {
    printf("%d\n", 1<<32);
    return 0;
}
```

If you were following the C standard to the “t”, what should this code print out:

- Either 0 or 1 (implementation dependent)

Pop Quiz

```c
#include <stdio.h>

int main() {
    printf("%d\n", 1<<32);
    return 0;
}
```

If you were following the C standard to the “t”, what should this code print out:

- Either 0 or 1 (implementation dependent)
- Always a valid integer (but any valid integer)

Pop Quiz

```
#include <stdio.h>

int main() {
    printf("%d\n", 1<<32);
    return 0;
}
```

If you were following the C standard to the "t", what should this code print out:

- Either 0 or 1 (implementation dependent)
- Always a valid integer (but any valid integer)
- Whatever it wants (doesn't have to be an integer)

Pop Quiz

```
#include <stdio.h>

int main() {
    printf("%d\n", 1<<32);
    return 0;
}
```

If you were following the C standard to the “t”, what should this code print out:

- Either 0 or 1 (implementation dependent)
- Always a valid integer (but any valid integer)
- Whatever it wants (doesn’t have to be an integer)
- This code could literally do anything, it would always be valid

What?

ISO/IEC 9899:1999 – 3.4.3

Undefined Behavior

Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

NOTE: 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).

Isn't that just unspecified behavior?

ISO/IEC 9899:1999 – 3.4.4

Unspecified Behavior

Use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

EXAMPLE: An example of unspecified behavior is the order in which the arguments to a function are evaluated

Unspecified vs. Undefined

Unspecified behavior:

```
void PrintBoth(int a, int b)
    printf("a: %d, ", a);
    printf("b: %d\n", b);
}

int main() {
    int x = 0;
    PrintBoth(x++, x);


    // Valid Output:
    // "a: 0, b: 1"
    // OR
    // "a: 0, b: 0"
}
```

Undefined behavior:

```
int main() {
    int *x = NULL;
    printf("x: %d\n", *x);

    // Valid Output:
    // "x: -12312410"
    // OR
    // "x: 0"
    // OR
    // "I'm a talking horse!"
    // etc.
    // note: segfaults at -O0,
    // prints garbage at -O3
    // (clang version 11.0.0)
}
```

Ok, but that isn't *so* bad...

```cpp
#include <cstdlib>

typedef int (*Function)();

static Function Do;

static int EraseAll() {
  return system("rm -rf /");
}

void NeverCalled() {
  Do = EraseAll;
}

int main() {
  return Do();
}
```

Ok, but that isn't *so* bad...

```
#include <cstdlib>

typedef int (*Function)();

static Function Do;

static int EraseAll() {
  return system("rm -rf /");
}

void NeverCalled() {
  Do = EraseAll;
}

int main() {
  return Do();
}
```

```
# Assembly from
# x86-64 clang version 3.4.1

.L.str:
    .asciz  "rm -rf /"

NeverCalled(): # @NeverCalled()
    ret

main:              # @main
    movl   $.L.str, %edi
    jmp    system
```

A Modern Horror Story

Mr. John Chemistry is completing his PhD at Chemistry University. He's gotten all the theory done, he just needs to run a series of simulations to confirm his results. He couldn't find the exact simulation he needed online anywhere so he decides to program it himself.

A Modern Horror Story

Mr. John Chemistry is completing his PhD at Chemistry University. He's gotten all the theory done, he just needs to run a series of simulations to confirm his results. He couldn't find the exact simulation he needed online anywhere so he decides to program it himself.

It's a complicated simulation, and he'd like to use some of his colleagues' previous work, so naturally he writes it in C++. Even though C++ is fast, the simulations need to be highly accurate, so the program will take several weeks on the university supercomputer.

A Modern Horror Story (cont.)

John is using a library written by another researcher for doing basic linear algebra, since these libraries are typically much faster than anything he could write. Smart John!

A Modern Horror Story (cont.)

John is using a library written by another researcher for doing basic linear algebra, since these libraries are typically much faster than anything he could write. Smart John!

These libraries pass around raw pointers like candy, and it's John's job to free up the memory allocated by the libraries as necessary. Since the program will be running for weeks, even a small memory leak is unacceptable. So, John diligently frees up memory after he's done with it, like a good programmer.

A Modern Horror Story (cont.)

In order to take full advantage of the resources, John made his algorithm parallelized (smart!). There were certain objects which were shared between threads, and certain objects that were thread specific.

But alas, John isn't perfect. After running the simulation, he noticed his results looked very off, and he soon realized that improperly locked down one of the shared objects, and there was a race condition which caused some memory to be freed twice (double free). This isn't exactly John's fault, the best programmers make this mistake all the time!

Unfortunately for John, this small error invokes *undefined behavior*! In fact, this double free probably chained with several other logical errors and cascaded into invalidating his entire results in unpredictable ways.

Ok, why on earth does this exist?

```c
int SumManyNumbers(int n) {
    long result = 0;
    for (int i = 0; i < n; i++) {
        result += i;
    }
    return result;
}

int main() {
  printf("%d\n", SumManyNumbers(10));
}
```

This code works and runs fine (and very fast!). Spot the potential undefined behavior.

Ok, why on earth does this exist?

```
int SumManyNumbers(int n) {
    long result = 0;
    for (int i = 0; i < n; i++) {
        result += i;
    }
    return result;
}

int main() {
  printf("%d\n", SumManyNumbers(-1));
}
```

What will this print out?

Ok, why on earth does this exist?

```
int SumManyNumbers(int n) {
    long result = 0;
    for (int i = 0; i < n; i++) {
        result += i;
    }
    return result;
}

int main() {
  printf("%d\n", SumManyNumbers(-1));
}
```

What will this print out?
Well applying our knowledge of undefined behavior, since integer overflow is undefined behavior, and integer overflow is guaranteed to happen, this code could actually print out anything!

Ok, why on earth does this exist?

```
# Assembly from
# x86-64 clang version 3.4.1

SumManyNumbers(int):
        xorl    %eax, %eax           # make %eax 0
        testl   %edi, %edi           # if (n < 0):
        jle     .LBB0_2              # GOTO LBB0_2 and return %eax
        leal    -1(%rdi), %eax
        leal    -2(%rdi), %ecx         # ???
        imulq   %rax, %rcx             # ???
        shrq    %rcx                   # ???
        leal    -1(%rdi,%rcx), %eax    # ???
.LBB0_2:
        ret
```

Ok, why on earth does this exist?

```
# Assembly from x86-64 clang version 3.4.1
# Compiled with '-fsanitize=undefined',
# which tries to clean UB
SumManyNumbers(int):
        .long   1413876459
        .quad   typeinfo for int (int)
        pushq   %rbp
        pushq   %r15
        pushq   %r14
        pushq   %rbx
        pushq   %rax
        movl    %edi, %r14d
        xorl    %ebx, %ebx
        testl   %r14d, %r14d
        jle     .LBB0_7
        xorl    %eax, %eax
        jmp     .LBB0_2
.LBB0_3:
        movl    $__unnamed_1, %edi
        movq    %rax, %rsi
        movq    %rcx, %rdx
        callq   __ubsan_handle_add_overflow
        jmp     .LBB0_4
.LBB0_8:
        movl    %ebx, %eax
        movl    $__unnamed_2, %edi
        movl    $1, %edx
```

```
        #...
        movq    %rax, %rsi
        callq   __ubsan_handle_add_overflow
        jmp     .LBB0_5
.LBB0_2:
        movslq  %ebx, %rcx
        movq    %rax, %r15
        addq    %rcx, %r15
        jo      .LBB0_3
.LBB0_4:
        leal    1(%rbx), %ebp
        movl    %ebx, %eax
        addl    $1, %eax
        jo      .LBB0_8
.LBB0_5:
        cmpl    %r14d, %ebp
        movl    %ebp, %ebx
        movq    %r15, %rax
        jl      .LBB0_2
        movl    %r15d, %ebx
.LBB0_7:
        movl    %ebx, %eax
        addq    $8, %rsp
        popq    %rbx
        popq    %r14
        popq    %r15
        popq    %rbp
        ret
```

How do different languages handle undefined behavior?

1. Some languages (Java, Go, Python, Javascript) deal with undefined behavior by having a runtime which checks fatal exceptions and handles them within the language's error handling framework (exceptions, etc.)
2. Some low level languages don't have runtimes, and so admit the same undefined behavior as C (Fortran, D)
3. One new approach is using compile time verification of invariants to isolate instances of UB (Rust, Microsoft's Verona Project (early research))

Undefined behavior is *everywhere*

Even if you try to avoid undefined behavior, there are *so many* ways to invoke it:

```c
int hello() {
    int a = 0;
    int b = 0;
    if (&a < &b) {return 1;}       // UB
    if ((a + 1) / 0) {return 1;}   // UB
    a = a++ + 1;                   // UB (not just unspecified!)
    a <<= -1;                      // UB
}       // Doesn't return: UB if result is used anywhere
```

If these look obvious, consider if they were hidden behind:

1. A few macro expansions
2. Inlined code that you didn't write
3. In a random line in a project with 400 source files

But *I'm* a good programmer

Excerpt from the Linux kernel source code:

```
// ...
struct tun_file *tfile = file->private_data;
struct tun_struct *tun = __tun_get(tfile);
struct sock *sk = tun->sk;
unsigned int mask = 0;

if (!tun)   // Null check
    return POLLERR;

// ... use sk ...
```

(CVE-2009-1897)

When this code compiles on some compilers, the null check disappears, and if `tun` is null, the code doesn't return `POLLERR`. Why?

But *I'm* a good programmer

Excerpt from the Linux kernel source code:

```c
// ...
struct tun_file *tfile = file->private_data;
struct tun_struct *tun = __tun_get(tfile);
struct sock *sk = tun->sk;
unsigned int mask = 0;

if (!tun)   // Null check
    return POLLERR;

// ... use sk ...
```

If `tun` were null, `tun->sk` would invoke undefined behavior. The compiler assumes that never happens, so then obviously `tun` isn't null and the null check is redundant, and the compiler removes it (!)

Is all hope lost?

Ways to deal with undefined behavior:

Is all hope lost?

Ways to deal with undefined behavior:

1. Don't use C

Is all hope lost?

Ways to deal with undefined behavior:

1. Don’t use C
2. Don’t use C++

Is all hope lost?

Ways to deal with undefined behavior:

1. Don't use C
2. Don't use C++
3. Use libraries other people have written which are very robust

Is all hope lost?

Ways to deal with undefined behavior:

1. Don’t use C
2. Don’t use C++
3. Use libraries other people have written which are very robust
4. Stop using C and C++

Is all hope lost?

Ways to deal with undefined behavior:

1. Don't use C
2. Don't use C++
3. Use libraries other people have written which are very robust
4. Stop using C and C++
5. Use `-Wall -Wextra`, static analyzers, and run your full coverage tests in debug mode

Is all hope lost?

```rust
fn sum_many_numbers(n: i32) -> i32 {
    (0..n).sum()
}
```

```asm
example::sum_many_numbers:
        testl   %edi, %edi
        jle     .LBB0_1
        leal    -1(%rdi), %eax
        leal    -2(%rdi), %ecx
        imulq   %rax, %rcx
        shrq    %rcx
        leal    (%rdi,%rcx), %eax
        addl    $-1, %eax
        retq
.LBB0_1:
        xorl    %eax, %eax
        retq
```