Undefined Behavior Is Really Undefined

Nov 30, 2018 • Thomas Pornin

Undefined Behavior is the expression used in C and C++ to describe a situation in which, basically, “anything goes”. Historically, UB covered cases where existing C compilers (and architectures) would act in irreconcilable ways, and the standard committee, in its infinite wisdom, decided not to decide (that is, not to rule out any of the competing implementations). UB was also applicable to possible scenarios in which the standard, while usually quite exhaustive, would have failed to define a prescribed behavior. A third and increasingly more important meaning of UB is the following: UB represents opportunities for optimizations. And C and C++ developers really *love* optimizations; they demand with great insistence that compilers do their best to make the code fast.

Here is a classic example. Consider the following code:

```
void
foo(double *src, int *dst)
{
    int i;

    for (i = 0; i < 4; i ++) {
        dst[i] = (int)src[i];
    }
}
```

Compile this code on a 64-bit x86 platform, running Linux, with GCC (it’s an up-to-date Ubuntu 18.04, GCC version is 7.3.0). We want full optimization, and then have a look at the assembly output, hence the options are “`-W -Wall -O9 -S`” (the “`-O9`” argument selects the maximum optimization level of GCC, which is in practice equivalent to “`-O3`”, although there have been some GCC forks that defined higher levels). Here is the result:

```
        .file   "zap.c"
        .text
        .p2align 4,,15
        .globl  foo
        .type   foo, @function
foo:
.LFB0:
        .cfi_startproc
        movupd  (%rdi), %xmm0
        movupd  16(%rdi), %xmm1
        cvttpd2dq       %xmm0, %xmm0
        cvttpd2dq       %xmm1, %xmm1
        punpcklqdq      %xmm1, %xmm0
        movups  %xmm0, (%rsi)
        ret
        .cfi_endproc
.LFE0:
        .size   foo, .-foo
        .ident  "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
        .section        .note.GNU-stack,"",@progbits
```

Each of the first two `movupd` opcodes reads two `double` values into a 128-bit SSE2 register (a `double` is 64 bits, an SSE2 register can contain two of them). In other words, the four source values are read, and only then the conversions to `int` occur (the `cvttpd2dq` opcodes). The `punpcklqdq` moves around the four resulting 32-bit integers into a single SSE2 register (`%xmm0`), which is then written into RAM (`movups`). Notice the important bit: this C program nominally requests the following sequence of accesses to RAM:

- Read the first `double` value at `src[0]` .
- Write the first `int` value into `dst[0]` .
- Read the second `double` value at `src[1]` .
- Write the second `int` value into `dst[1]` .
- Read the third `double` value at `src[2]` .
- Write the third `int` value into `dst[2]` .
- Read the fourth `double` value at `src[3]` .
- Write the fourth `int` value into `dst[3]` .

However, what the C program asks for is only within the context of the *abstract machine* defined by the C standard; how things happen on the physical machine may differ. The compiler is free to reorder or modify the operations as long as the result still matches the semantics of the abstract machine (it is known as the "as-if rule"). Notably, in this case, the compiled code does things in a different order:

- Read the first `double` value at `src[0]` .
- Read the second `double` value at `src[1]` .
- Read the third `double` value at `src[2]` .
- Read the fourth `double` value at `src[3]` .
- Write the first `int` value into `dst[0]` .
- Write the second `int` value into `dst[1]` .
- Write the third `int` value into `dst[2]` .
- Write the fourth `int` value into `dst[3]` .

This is C: everything in memory is ultimately *bytes* (i.e. `unsigned char` slots, in practice octets), and arbitrary pointer arithmetic is supported. In particular, the `src` and `dst` pointers might be made by the caller to point to memory areas that overlap in some way (this is known as "aliasing"). Thus, the order of reads and writes may matter, in case some bytes are written to then read back again. To fully conform to the abstract behavior defined by the C program, the C compiler would have to interleave the reads and writes, and make the code do a full round-trip to RAM at each loop iteration. The resulting code would be larger, and much slower. C developers would wail and whine.

Fortunately, *undefined behavior* comes to the rescue. The C standard specifies that values "cannot" be accessed through pointers that do not match the effective type of the value; in plain words, that if a value is written into `dst[0]`, with `dst` being a pointer to `int`, the corresponding bytes cannot be read back with `src[1]`, where `src` is a pointer to `double`, because that would be accessing what has become an `int` value through a pointer to an incompatible type. Such an access would imply undefined behavior. In the ISO 9899:1999 standard (aka "C99"), this is specified in section 6.5, subclause 7 (in the more recent revision 9899:2018, aka "C17", the wording is unchanged); this is also known as *strict aliasing*. The consequence here is that the C compiler is allowed to *assume* that accesses that imply UB by breaking aliasing rules do not happen; thus, the compiler can reorder the reads and writes in any way it wants, since they should not overlap. This is how the code is optimized.

This is in a nutshell what UB means: the compiler can assume that UB does not happen, and produce code under that assumption. In the case of strict aliasing rules, UB allows for important optimizations to take place, that would be hard to obtain otherwise, as long as aliasing can occur. More generally, in the code generation routines used by the compiler, each instruction has dependencies that restrict the opcode scheduling algorithm: an instruction cannot be issued before the instructions that it depends upon, or after the instructions that depend on it. In the example above, UB removes the dependencies between the writes to `dst[]`, and the "subsequent" reads from `src[]`: such a dependency can exist only if an UB-implying access exists. In a similar way, the notion of UB can allow the compiler to simply *remove* code that cannot happen without going through an UB condition.

This is all well and dandy, but the same behavior is also occasionally felt as the compiler backstabbing the developer. A commonly encountered comment is that "the compiler uses UB as a pretext to crash my code". Suppose that somebody is writing code that adds integer values together, and fears overflows; such things have happened in Bitcoin. The developer might think along the following lines: the CPU uses two's complement representation of integers, and therefore, if an overflow occurs, this will be because the result is truncated to the type width, e.g. 32 bits. This means that the result of an overflow is predictable, and can be tested. Our developer will write this:

```
#include <stdio.h>
#include <stdlib.h>

int
add(int x, int y, int *z)
{
    int r = x + y;
    if (x > 0 && y > 0 && r < x) {
        return 0;
    }
    if (x < 0 && y < 0 && r > x) {
        return 0;
    }
    *z = r;
    return 1;
}

int
main(int argc, char *argv[])
{
    int x, y, z;
    if (argc != 3) {
        return EXIT_FAILURE;
    }
    x = atoi(argv[1]);
    y = atoi(argv[2]);
    if (add(x, y, &z)) {
        printf("%d\n", z);
    } else {
        printf("overflow!\n");
    }
    return 0;
}
```

Now let’s try to compile that with GCC:

```
$ gcc -W -Wall -O9 testadd.c
$ ./a.out 17 42
59
$ ./a.out 2000000000 1500000000
overflow!
```

OK, this seems to work. Let’s try again with another compiler, Clang (version 6.0.0 on my system):

```
$ clang -W -Wall -O3 testadd.c
$ ./a.out 17 42
59
$ ./a.out 2000000000 1500000000
-794967296
```

Wut?

It turns out that when an operation on signed integer types has a result which is not representable in the target type, we enter UB territory. But the compiler can assume that UB does not happen. In particular, when optimizing the expression “`x > 0 && y > 0 && r < x`”, the compiler infers that since `x` and `y` are strictly positive, the third test cannot be true (their sum cannot be lower than either), and the whole thing can be skipped. In other words, the overflow being UB, it “cannot happen" from the compiler perspective, and all instructions that depend upon such a condition can be removed. The overflow detection mechanism has simply disappeared.

The assumption that computations use "wraparound semantics" (as the CPU opcodes actually do) has never been standard for signed types; it was *traditional*, at a time when compilers were not smart enough to optimize code based on range analysis. It is possible to force Clang and GCC to enforce wraparound semantics on signed types with the special flag `-fwrapv` (on Microsoft Visual C, you might use `-d2UndefIntOverflow-`, as described in this blog entry). This is however relatively fragile, and that flag may disappear when the code is reused in another project or on another architecture.

That overflows on signed types imply UB is not well-known. In the C99 and C17 standards, it is described in section 6.5, subclause 5:

> If an *exceptional* condition occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

However, for unsigned types, modular semantics are guaranteed. Section 6.2.5, subclause 9, specifies that:

> A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

Another example of UB with signed types happens with divisions. It is well-known that division by zero is not mathematically defined, and thus dividing by zero leads to undefined behavior, as per the standard. On x86 CPU, if the `idiv` opcode is invoked with a denominator equal to zero, a CPU exception happens. CPU exceptions are like interrupt requests, they are handled by the operating system; on Unix-like systems such as Linux, the CPU exception triggered by `idiv` is translated into a `SIGFPE` signal delivered to the process, and the default handler kills the process (nevermind that "FPE" means "floating-point exception", while `idiv` works with integers). However, there is another situation that also implies UB. Consider this code:

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    int x, y;
    if (argc != 3) {
        return EXIT_FAILURE;
    }
    x = atoi(argv[1]);
    y = atoi(argv[2]);
    printf("%d\n", x / y);
    return 0;
}
```

And then try it:

```
$ gcc -W -Wall -O testdiv.c
$ ./a.out 42 17
2
$ ./a.out -2147483648 -1
zsh: floating point exception (core dumped)  ./a.out -2147483648 -1
```

Indeed, on that machine (still my common x86 running Linux), the `int` type can represent values from -2147483648 to +2147483647. When dividing -2147483648 by -1, I should get +2147483648. But this does not fit in an `int`. Therefore, behavior is undefined. Anything can happen. In this case, the process dies. On other systems, you could get a different outcome, especially the small CPU that do not have a division opcode; on such architectures, division is done in software, with a routine that is normally provided by the compiler, and that one may do whatever it wants on UB, because that's exactly what UB means.

As a side-note, a `SIGFPE` can also be obtained under the same conditions with the modulo operator (`%`). Indeed, internally, the same `idiv` opcode is used to get both the quotient and remainder, hence the same CPU exception is triggered. Interestingly, as per the C99 standard, the "`INT_MIN % -1`" expression would not have invoked UB, because the mathematical result is well defined (it is zero) and it certainly fits into the representable range of the target type. In the C17 standard, the wording of 6.5.5, subclause 6, was augmented to cover that case and align the standard with the reality of common hardware platforms.

There are many subtle conditions that can lead to UB. Look at this code:

```
#include <stdio.h>
#include <stdlib.h>

unsigned short
mul(unsigned short x, unsigned short y)
{
    return x * y;
}

int
main(int argc, char *argv[])
{
    int x, y;
    if (argc != 3) {
        return EXIT_FAILURE;
    }
    x = atoi(argv[1]);
    y = atoi(argv[2]);
    printf("%d\n", mul(x, y));
    return 0;
}
```

As per the C standard, what is this C program allowed to print, if invoked with parameters “45000” and “50000”?

- A) 18048
- B) 2250000000
- C) God save the Queen

And the answer is… all of the above! You might have told yourself that since `unsigned short` is an unsigned type, it should work with wraparound semantics, modulo 65536 because on an x86 CPU, that type typically has size exactly 16 bits (as per the standard, it could be larger than that, but in practice it will be a 16-bit type). And while the product is mathematically 2250000000, it will be reduced modulo 65536, yielding 18048. *However*, this would be forgetting about the *integer promotions*. As per the C standard (section 6.3.1.1, subclause 2), when the operands are of a type strictly smaller than `int`, and all values of that type can be represented without loss in an `int` (which is our case here: on my x86 running Linux, an `int` is 32 bits and can certainly hold all values from 0 to 65535), then both operands are converted to `int`, and the operation occurs on `int` values. In particular, the product is computed as an `int`, and only *coerced* back to the `unsigned short` type when returning from the function (this is when the modulo 65536 occurs). The problem is that the mathematical result before that coercion is 2250000000, and that value does not fit in an `int`, which is a signed type. Hence UB. Therefore, anything can happen afterwards, including a spontaneous display of British patriotism.

In practice, though, you will get 18048 with usual compilers, because there is no currently known and implemented optimization that could leverage the UB to optimize things *in this specific program* (one can imagine other more contrived scenarios where that UB really wreaks havoc to execution).

A final example, this one in C++:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <array>

int
main(int argc, char *argv[])
{
    std::array<char, 16> tmp;
    int i;

    if (argc < 2) {
        return EXIT_FAILURE;
    }
    memset(tmp.data(), 0, 16);
    if (strlen(argv[1]) < 16) {
        strcpy(tmp.data(), argv[1]);
    }
    for (i = 0; i < 17; i ++) {
        printf(" %02x", tmp[i]);
    }
    printf("\n");
}
```

This is *not* a classic "`strcpy()` is bad bad bad" example. Indeed, in the code above, the `strcpy()` is performed only if the source string is small enough, including the terminating zero. Moreover, the array contents are explicitly initialized to zero, so all array bytes have a set value, whether the argument string is large or not. However, the final loop is incorrect, in that it *reads* one more byte than it should.

Let's try it:

```
$ g++ -W -Wall -O9 testvec.c
$ ./a.out foo
 66 6f 6f 00 00 00 00 00 00 00 00 00 00 00 00 00 10 58 ffffffca ff
ffffac ffffffc0 55 00 00 00 ffffff80 71 34 ffffff99 07 ffffffba ff
ffffea ffffffd0 ffffffe5 44 ffffff83 fffffffd 7f 00 00 00 00 00 00
 00 00 00 00 10 58 ffffffca ffffffac ffffffc0 55 00 00 ffffff97 7b
 12 1b ffffffa1 7f 00 00 02 00 00 00 00 00 00 00 ffffffd8 ffffffe5
 44 ffffff83 fffffffd 7f 00 00 00 ffffff80 00 00 02 00 00 00 60 56
(...)
62 64 3d 30 30
zsh: segmentation fault (core dumped)  ./a.out foo
```

Wut++?

Naively, you could say: OK, it is reading one byte out of bounds; but that should be no big deal, that byte is still on the stack, memory is mapped, there should be no issue except an extra seventeenth number with a somewhat unpredictable value. The final loop will still print exactly 17 integers (in hexadecimal) and exit cleanly.

But the compiler sees things differently. It knows perfectly well that the seventeenth access invokes UB. Within the logic of the compiler, this means that any instruction beyond that one is then in limbo: things after UB are not required to exist (technically, even *previous* instructions can be affected, because UB can be retroactive). In particular, it will simply skip the condition test on the loop, and the loop will run forever; or, more accurately, until the reading goes beyond the pages allocated for the stack, at which point a `SIGSEGV` occurs.

Amusingly, if GCC is invoked with less aggressive optimizations, it will warn in a quite explicit way:

```
$ g++ -W -Wall -O1 testvec.c
testvec.c: In function ‘int main(int, char**)’:
testvec.c:20:15: warning: iteration 16 invokes undefined behavior [-Wagg
         printf(" %02x", tmp[i]);
         ~~~~~~^~~~~~~~~~~~~~~~
testvec.c:19:19: note: within this loop
     for (i = 0; i < 17; i ++) {
                 ~~^~~~
```

But at `-O9` level, somehow, the warning disappears. This may be related to the more forcefull loop unrolling that happens at high optimization levels. Arguably, this is a GCC bug (the lack of warning, that is; as per the standard, what GCC does is fully conformant either way, and no "diagnostic" is required by the standard in that situation).

Conclusion: if you write code in C or C++, take *great care* that you always avoid undefined behavior conditions, even in situations that seem "obviously harmless". Using unsigned integer types helps with arithmetic computations, because they have guaranteed modular semantics (but you can still run into trouble with integer promotions). Another possible stance, but strangely unpopular, might be: don't write code in C or C++. This is not always an option, for a variety of reasons. But in situations where you *can* choose between languages, e.g. when starting a new project on a platform where Go or Rust or Java or other languages are supported, then it can pay off to not necessarily go for C as the "default language". All choices of tools, including programming languages, imply trade-offs; the pitfalls of C programming, notably UB conditions on arithmetics on signed types, tend to imply maintenance efforts in the long run that are often underestimated.