A Guide to Undefined Behavior in C and C++, Part 3

Also see Part 1 and Part 2.

A C or C++ implementation must perform side effecting operations in program order. For example, if a program executes these lines of code:

```
printf ("Hello\n")
printf ("world.\n");
```

It is impermissible for the implementation to print:

```
world.
Hello
```

This is obvious, but keep in mind that other things your program does can be freely reordered by the compiler. For example if a program increments some variables in one order:

```
a++;
b++;
```

a compiler could emit code incrementing them in the other order:

```
incl      b
incl      a
```

The optimizer performs transformations like this when they are thought to increase performance and when they do not change the program's observable behavior. In C/C++, observable behaviors are those that are side effecting. You might object, saying that this reordering is in fact observable. Of course it is: if a and b are globals, a memory-mapped I/O device, a signal handler, an interrupt handler, or another thread could see the memory state where b but not a has been written. Nevertheless, the reordering is legal since stores to global variables are not defined as side-effecting in these languages. (Actually, just to make things

confusing, stores to globals are side effecting according to the standard, but no real compiler treats them as such. See the note at the bottom of this post.)

With this background material out of the way, we can ask: Is a segfault or other crash a side effecting operation in C and C++? More specifically, when a program dies due to performing an illegal operation such as dividing by zero or dereferencing a null pointer, is this considered to be side effecting? The answer is definitely "no."

An Example

Since crash-inducing operations are not side effecting, the compiler can reorder them with respect to other operations, just like the stores to a and b in the example above were reordered. But do real compilers take advantage of this freedom? They do. Consider this code:

```
volatile int r;

int s;

void foo1 (unsigned y, unsigned z, unsigned w)
{
  r = 0;
  s = (y%z)/w;
}
```

foo1 first stores to r; this store is a side-effecting operation since r is volatile. Then foo1 evaluates an expression and stores the result to s. The right hand side of the second assignment has the potential to execute an operation with undefined behavior: dividing by zero. On most systems this crashes the program with a math exception.

Here's what a recent version of GCC outputs:

```
foo1:
  pushl     %ebp
  xorl       %edx, %edx
  movl       %esp, %ebp
  movl       8(%ebp), %eax
  divl       12(%ebp)          <-- might crash the program
  movl       $0, r             <-- side effecting operation
  movl       %edx, %eax
  xorl       %edx, %edx
  divl       16(%ebp)
  popl       %ebp
  movl       %eax, s
  ret
```

Notice that a divl instruction — which may kill the program if the divisor is zero — precedes the store to r. Clang's output is similar:

```
foo1:
  pushl     %ebp
  movl       %esp, %ebp
  movl       8(%ebp), %eax
  xorl       %edx, %edx
  divl       12(%ebp)          <-- might crash the program
  movl       %edx, %eax
  xorl       %edx, %edx
  divl       16(%ebp)
  movl       $0, r             <-- side effecting operation
  movl       %eax, s
  popl       %ebp
  ret
```

And Intel CC:

```
foo1:
  movl          8(%esp), %ecx
  movl          4(%esp), %eax
  xorl          %edx, %edx
  divl          %ecx              <-- might crash the program
  movl          12(%esp), %ecx
  movl          $0, r             <-- side effecting operation
  movl          %edx, %eax
  xorl          %edx, %edx
```

```
  divl          %ecx
  movl          %eax, s
  ret
```

In compiler speak, the problem here is that there are no dependencies between the first and second lines of code in foo1. Therefore, the compiler can do whatever kinds of reordering seem like a good idea to it. Of course, in the bigger picture, there is a dependency since the computation involving r will never get to execute if the RHS of the assignment into s crashes the program. But — as usual — the C/C++ standard creates the rules and the compiler is just playing by them.

Another Example

This C code calls an external function and then does a bit of math:

```
void bar (void);

int a;

void foo3 (unsigned y, unsigned z)
{
  bar();
  a = y%z;
}
```

Clang does the math before calling the function:

```
foo3:
  pushl     %ebp
  movl       %esp, %ebp
  pushl     %esi
  subl       $4, %esp
  movl       8(%ebp), %eax
  xorl       %edx, %edx
  divl       12(%ebp)          <-- might crash the program
  movl       %edx, %esi
  call       bar               <-- might be side effecting
  movl       %esi, a
  addl       $4, %esp
  popl       %esi
  popl       %ebp
  ret
```

To make a complete program we can add this code:

```
void bar (void)
{
  setlinebuf(stdout);
  printf ("hello!\n");
}
```

```
int main (void)
{
  foo3(1,0);
  return 0;
}
```

Now we can compile and run it:

```
regehr@john-home:~$ clang -O0 biz_main.c biz.c -o biz
regehr@john-home:~$ ./biz
hello!
Floating point exception
regehr@john-home:~$ clang -O1 biz_main.c biz.c -o biz
regehr@john-home:~$ ./biz
Floating point exception
```

As we predicted from looking at the assembly code for foo3(), when optimization is turned on the division instruction crashes the program before the unbuffered printf() can occur.

Why does this matter?

Hopefully the previous example made the relevance obvious, but here I'll spell it out. Let's say you're debugging a difficult segfault:

1. You add a printf() just before calling foo(), where foo() is a nasty piece of logic that contains some suspicious pointer operations. Of course — as we did in the example — you turn off buffering on stdout so that lines are actually pushed to the terminal before the program continues to execute.
2. You run the code and see that the printf() is not reached before the program segfaults.
3. You conclude that the crash is not due to foo(): it must have occurred in earlier code.

The inference in step 3 is wrong if some dangerous operation in foo() was moved in front of the printf() and then triggered a segfault. When you make an incorrect inference while debugging, you start walking down the wrong path and can waste a lot of time there.

Solutions

If you are experiencing this kind of problem, or are just paranoid about it, what should you do? The obvious things to try are turning off optimizations or single-stepping through your code's instructions. For embedded system developers, neither of these may be feasible.

A solution that failed in a surprising way is adding a compiler barrier. That is, changing foo1 to read:

```
void foo2 (unsigned y, unsigned z, unsigned w)
{
  r = 0;
  asm volatile ("" : : : "memory");
  s = (y%z)/w;
}
```

The new line of code is a GCC idiom (also understood by Intel CC and Clang) which means "this inline asm, although it contains no instructions, may touch all of memory." The effect is basically to artificially inject a lot of dependencies into the code, forcing all register values to be saved to RAM before the asm and to be reloaded afterward. This new code causes Intel CC and GCC to store to r before evaluating any of the RHS of s, but Clang goes ahead and emits this code:

```
foo1:
  pushl     %ebp
  movl       %esp, %ebp
  movl       8(%ebp), %eax
  xorl       %edx, %edx
  divl       12(%ebp)          <-- might crash the program
  movl       %edx, %eax
  xorl       %edx, %edx
```

```
divl    16(%ebp)
movl    $0, r           <-- side effecting operation
movl    %eax, s
popl    %ebp
ret
```

I believe that Clang is operating correctly. The compiler barrier adds some artificial dependencies, but not enough of them to stop a divide instruction from moving ahead of the store-to-volatile.

The real solution to this problem — which is useful to understand, although it does us no good as long as we're stuck with C and C++ — is to assign a semantics to exceptional situations such as divide by zero and dereferencing a null pointer. Once a language does this, it becomes possible to constrain the optimizer's behavior with respect to reordering operations that are potentially dangerous. Java does a fine job at this.

A Technical Note on Side Effects in C and C++

The C99 standard states that:

Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment.

In the C standard "object" means "variable." The C++0x draft contains very similar language. The "modifying an object" clause is nonsensical because — in combination with sequence point semantics — it forbids the compiler from eliminating the redundant store when a program contains code like this:

```
x=0;
x=0;
```

No optimizing C or C++ compiler considers "modifying an object" to be a side-effecting behavior.

While we're nitpicking, the "calling a function that does any of those operations" bit is also widely ignored, due to function inlining. The code inside the inlined function may be side effecting, but the call itself is never treated as such.

Update from August 19, 2010

CompCert is the most practical verified C compiler anyone has yet developed, it is an amazing piece of work. I spent some time trying to get it to produce object code of the kind that I've shown above — where a dangerous operation is moved in front of a side-effecting operation — and was unable to. Xavier Leroy, the CompCert project lead, had this to say:

> Actually, you're putting your finger on a relatively deep semantic issue.
>
> We all agree that C compilers do not have to preserve undefined behaviors: a source program that accesses an array out of bounds doesn't always crash when run. One can regret it (many security holes would be avoided if it were so), but we sort of agree that the cost of preserving undefined behaviors would be too high (array bound checks, etc).
>
> Now, a compiler that preserves defined behaviors has two choices:
>
> 1- If the source program exhibits an undefined behavior, the compiled code can do absolutely whatever it pleases.
>
> 2- If the source program exhibits an undefined behavior, the compiled code executes like the source program up to the undefined behavior point, with the same observable effects, then is free to do whatever it wants (crashing or continuing with arbitrary effects).

If I read your post correctly, you've observed choice 1 and point out that choice 2 would be not only more intuitive, but also more helpful for debugging. I think I agree with you but some compiler writers really like choice 1...

For CompCert, the high-level correctness theorems I proved are all of the first kind above: if the source program goes wrong, nothing is guaranteed about the behavior of the compiled code. It happens, however, that the stepwise simulation lemmas I build on actually prove property 2 above: the compiled code cannot crash "earlier" than the source code, and will produce the same pre-crash observables, then possibly more. So maybe I should strengthen my high-level correctness theorems...

Concerning what's observable, I agree with you that the C standard talks nonsense: writes to nonvolatile objects are of course not observable, neither are function calls by themselves (only the system calls the function might invoke).

So there we have it: CompCert is provably immune to these problems, but in a way that is perhaps hacky. In a subsequent email Xavier mentioned that he's thinking about strengthening the top-level CompCert theorems to show this cannot happen. Cool!

July 30, 2010 regehr Computer Science, Software Correctness

---

24 responses to “A Guide to Undefined Behavior in C and C++, Part 3”

1. **Ben L. Titzer** says:
   August 3, 2010 at 12:19 pm

   It’s time for a better language for systems programming! 🙂

2. **regehr** says:
   August 3, 2010 at 12:24 pm

   Right! Got anything in mind? 🙂

3. **yoghurt** says:
   August 13, 2010 at 7:07 am

@regehr

Common Lisp, of course!

4. **Anonymous** says:
   August 18, 2010 at 4:41 am

   This just goes to show that printf() debugging (esp. of the "got here" kind) is no substitute for being aware of the edge cases of the various operators. Kind of obvious really: the programmer's substandard understanding of the language at hand results in an incorrect program, film at 11…

   However, these aren't sufficient reasons for constraining the compiler. Incorrect programs should not have a surrogate behaviour, and giving them one won't help in writing programs that are actually correct.

5. **Martin** says:
   August 19, 2010 at 3:45 pm

   I take exception to the programmers-substandard-understanding-of-the-language bit there. That puts the blame on programmers, when IMO (at least as far as C++ goes) is that it has almost simply gotten too hard to get it right… We need better compilers to warn us when we're nearing undefined behavior…

6. **regehr** says:
   August 19, 2010 at 9:53 pm

Martin, I totally agree with you. Anonymous, your position seems to be that language design is independent of usability, and that is simply not the case.

7. **ouah** says:
   October 12, 2010 at 6:55 am

   For me this reordering thing when there is a division and so a potential floating point exception looks like a compiler bug.

   There is an open issue and a discussion in GCC Bugzilla:
   http://gcc.gnu.org/bugzilla/show_bug.cgi?id=29186

8. **Tennessee C-Veilleux** says:
   October 21, 2010 at 10:50 am

   While reading the first bits of this post about operation re-ordering, I was thinking to myself "I hope compiler re-ordering barriers are mentioned!". Then you do mention them. However, to my great surprise, you also show that a compiler which supports GNU extensions in its front-end still manages to produce re-ordered code. Whoa !

   Seeing how this construct is used in my own embedded code and in the Linux source code (amongst other places), how are we to trust future better-than-GCC or safer-than-GCC compilers for high-performance OS code if we can't even prevent re-ordering explicitly at the source-code level ? Let's not even talk about the treatment of the "volatile" keyword, which trumps all efforts at extrapolation between compilers.

Also, one of the most common mental-model assumptions amongst programmers is that operations like assignments happen in source-order. Even though one can know (as many of us do) that optimizers can re-order these statements, it is still highly counter-intuitive, especially when trying to review code. The infix syntax and "math-like" assignment syntax of the language messes with our brain when we're not careful.

More and more, I'm starting to wonder how safety-critical code can continue being written in C. I understand why Ada was used nearly exclusively for so long in aerospace and now Java is starting to gain much traction, because of both of those languages better-defined (although still imperfect) semantics.

9. **Paolo G. Giarrusso** says:
   January 12, 2011 at 8:32 pm

   > While weâ€™re nitpicking, the "calling a function that does any of those operations" bit is also widely ignored, due to function inlining. The code inside the inlined function may be side effecting, but the call itself is never treated as such.
   You do mean that this clause is ignored if the function is inlined, don't you? My understanding is that calls to non-inlineable functions and modifications to globals are never reordered, because such function can modify globals – that's what prevent lock acquisition and release from being reordered, which in turn avoids that operations are moved out of a critical section! I also would guess that the writers of the C standard define side effects as they do, because they think of this situation.

10. **Margit** says:
    February 15, 2011 at 11:11 am

    In place of r = 0 in top example,
    #define SEQUENCE_SET(x,z) \
    do { x= z; \
    (void)x; \
    while (0)

    Then do
    SEQUENCE_SET (r, 0);

    Any compliant compiler must respect this.

    Margit

11. **regehr** says:
    February 15, 2011 at 11:16 am

    Margit, I'm afraid that is not the case.

12. **Margit** says:
    February 15, 2011 at 11:54 am

    Indeed, the "compliant" is somewhat construed.
    However, the construct works with all,
    avaliable to me, compilers.

(Win, AIX, SUN, BSD, Linux, Cygwin, MingW)
In fact the costruct (void)x is used quite commenley to overcome warnings from unused parameters in a function.

Margit

13. **Margit** says:
    February 15, 2011 at 11:57 am

    Actually it IS defined by the standard that defines an expression as a sequence point (IMHO).

    Margit

14. **regehr** says:
    February 15, 2011 at 12:02 pm

    Hi Margit,

    On my x86-64 machine your code fails to stop Clang and Intel CC from moving the divide in front of the assignment. It does stop GCC from doing the reordering, but I believe this is a coincidence, not required by the standard.

    The point of this article is that undefined behavior is not constrained by sequence point semantics.

15. **Margit** says:

February 15, 2011 at 12:08 pm

Oh, but a sequence point DOES define subsequent semantics.
It is required to do so (IMHO).

Margit

16. **Margit** says:
    February 15, 2011 at 12:11 pm

    Sorry, I should have appended –
    I believe that the construct (void)x according to
    my understanding of the C (maybe also C++) standard implies
    a sequence point or am I mistaken?

    Margit

17. **regehr** says:
    February 15, 2011 at 12:15 pm

    Hi Margit,

    There are multiple sequence points in your code, but there is also a sequence point at the end of the original assignment r=0;

    It would be nice if the effect of undefined behaviors was constrained by sequence points, but this is not my reading of the standard. In any case, it

doesn't matter what I think since the compiler writers have already decided what to do. You could file a bug about this I doubt it will change their mind.

18. **Margit** says:
    February 15, 2011 at 12:21 pm

    There is NOT a sequence point with
    assignment r=0;
    That is the point.

    Margit

19. **regehr** says:
    February 15, 2011 at 12:27 pm

    Margit, there is a sequence point at the end of "r=0;". See Annex C of the C99 standard which says there is a sequence point following a full expression, including an expression statement, which is what this is.

20. **Margit** says:
    February 15, 2011 at 12:29 pm

    But we are not talking about C99!

    Margit

21. **regehr** says:
    February 15, 2011 at 12:31 pm

Hi Margit, I'm losing interest in this thread.

22. **Margit** says:
    February 15, 2011 at 12:48 pm

    Sorry to hear that. I was hoping to include my
    observartions in your post.

    To iterate, a may NOT reorder instructions when
    a sequence-point occurs.

    To my mind, your obervations for Clang/Intel represent a bug.

    Margit

23. **regehr** says:
    February 15, 2011 at 1:09 pm

    Hi Margit,

    The ANSI C (C89) standard says: "The end of a full expression is a sequence point."

    Anyway, the real reason I'm not that interested in pursuing this further is that the developers of several major C implementations have decided what to do. It's fine if you believe this is a bug, but I strongly doubt they would agree. The point of my post was to call this issue to the attention of C developers, since it's nasty.

24. **Margit** says:
    February 15, 2011 at 1:53 pm

    Indeed, and I concur with your explanation.
    Indeed it is nasty, and I do not believe that
    it is possible (within the C standard) to
    code this possibility.
    However my code stands and is used universally.

    Margit