Undefined behavior can result in time travel (among other things, but time travel is the funkiest)

devblogs.microsoft.com/oldnewthing/20140627-00

June 27, 2014

Raymond Chen

The C and C++ languages are notorious for the very large section of the map labeled *here be dragons*, or more formally, *undefined behavior*.

When undefined behavior is invoked, anything is possible. For example, a variable can be both true and false. John Regehr has a list of interesting examples, as well as some winners of the ensuing contest.

Consider the following function:

```
int table[4];
bool exists_in_table(int v)
{
    for (int i = 0; i <= 4; i++) {
        if (table[i] == v) return true;
    }
    return false;
}
```

What does this have to do with time travel, you ask? Hang on, impatient one.

First of all, you might notice the off-by-one error in the loop control. The result is that the function reads one past the end of the `table` array before giving up. A classical compiler wouldn't particularly care. It would just generate the code to read the out-of-bounds array element (despite the fact that doing so is a violation of the language rules), and it would return `true` if the memory one past the end of the array happened to match.

A post-classical compiler, on the other hand, might perform the following analysis:

- The first four times through the loop, the function might return `true`.
- When `i` is 4, the code performs undefined behavior. Since undefined behavior lets me do anything I want, I can totally ignore that case and proceed on the assumption that `i` is never 4. (If the assumption is violated, then something unpredictable happens, but that's okay, because undefined behavior grants me permission to be unpredictable.)

- The case where `i` is 5 never occurs, because in order to get there, I first have to get through the case where `i` is 4, which I have already assumed cannot happen.
- Therefore, all legal code paths return `true`.

As a result, a post-classical compiler can optimize the function to

```
bool exists_in_table(int v)
{
    return true;
}
```

Okay, so that's already kind of weird. A function got optimized to basically nothing due to undefined behavior. Note that even if the value isn't in the table (not even in the illegal-to-access fifth element), the function will *still return true*.

Now we can take this post-classical behavior one step further: Since the compiler can assume that undefined behavior never occurs (because if it did, then the compiler is allowed to do *anything it wants*), the compiler can use undefined behavior to guide optimizations.

```
int value_or_fallback(int *p)
{
 return p ? *p : 42;
}
```

The above function accepts a pointer to an integer and either returns the pointed-to value or (if the pointer is null) returns the fallback value 42. So far so good.

Let's add a line of debugging to the function.

```
int value_or_fallback(int *p)
{
 printf("The value of *p is %d\n", *p);
 return p ? *p : 42;
}
```

This new line introduces a bug: It dereferences the pointer `p` without checking if it is null. This tiny bug actually has wide-ranging consequences. A post-classical compiler will optimize the function to

```
int value_or_fallback(int *p)
{
 printf("The value of *p is %d\n", *p);
 return *p;
}
```

because it observes that the null pointer check is no longer needed: If the pointer were null, then the `printf` already engaged in undefined behavior, so the compiler is allowed to do anything in the case the pointer is null (including acting as if it weren't).

Okay, so that's not too surprising. That may even be an optimization you expect from a compiler. (For example, if the ternary operator was hidden inside a macro, you would have expected the compiler to remove the test that is provably false.)

But a post-classical compiler can now use this buggy function to start doing time travel.

```
void unwitting(bool door_is_open)
{
 if (door_is_open) {
  walk_on_in();
 } else {
  ring_bell();
  // wait for the door to open using the fallback value
  fallback = value_or_fallback(nullptr);
  wait_for_door_to_open(fallback);
 }
}
```

A post-classical compiler can optimize this entire function to

```
void unwitting(bool door_is_open)
{
 walk_on_in();
}
```

Huh?

The compiler observed that the call `value_or_fallback(nullptr)` invokes undefined behavior on all code paths. Propagating this analysis backward, the compiler then observes that if `door_is_open` is false, then the `else` branch invokes undefined behavior on all code paths. Therefore, *the entire `else` branch can be treated as unreachable*.[2]

Okay, now here comes the time travel:

```
void keep_checking_door()
{
 for (;;) {
  printf("Is the door open? ");
  fflush(stdout);
  char response;
  if (scanf("%c", &response) != 1) return;
  bool door_is_open = response == 'Y';
  unwitting(door_is_open);
 }
}
```

A post-modern compiler may propagate the analysis that "if `door_is_open` is false, then the behavior is undefined" and rewrite this function to

```
void keep_checking_door()
{
 for (;;) {
  printf("Is the door open? ");
  fflush(stdout);
  char response;
  if (scanf("%c", &response) != 1) return;
  bool door_is_open = response == 'Y';
  if (!door_is_open) abort();
  walk_on_in();
 }
}
```

Observe that even though the original code rang the bell before crashing, the rewritten function skips over ringing the bell and just crashes immediately. You might say that the compiler *went back in time and unrung the bell*.

This "going back in time" is possible even for objects with external visibility like files, because the standard allows for *anything at all* to happen when undefined behavior is encountered. And that includes hopping in a time machine and pretending you never called `fwrite`.

Even if you claim that the compiler is not allowed to perform time travel,[1] it's still possible to see earlier operations become undone. For example, it's possible that the undefined operation resulted in the file buffers being corrupted, so the data never actually got written. Even if the buffers were flushed, the undefined operation may have resulted in a call to `ftruncate` to logically remove the data you just wrote. Or it may have resulted in a `DeleteFile` to delete the file you thought you had created.

All of these behaviors have the same observable effect, namely that the earlier action appears not to have occurred. Whether they actually occurred and were reversed or never occurred at all is moot from a compiler-theoretic point of view.

The compiler may as well have propagated the effect of the undefined operation backward in time.

[1] For the record, the standard explicitly permits time travel in the face of undefined behavior:

> However, if any such execution contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

(Emphasis mine.)

[2] Another way of looking at this transformation is that the compiler saw that the `else` branch invokes undefined behavior on all code paths, so it rewrote the code as

```
void unwitting(bool door_is_open)
{
 if (door_is_open) {
  walk_on_in();
 } else {
  walk_on_in();
 }
}
```

taking advantage of the rule that undefined behavior allows anything to happen, so in this case, it decided that "anything" was "calling `walk_on_in` by mistake."

Bonus chatter: Note that there are some categories of undefined behavior which may not be obvious. For example, dereferencing a null pointer is undefined behavior *even if you try to counteract the dereference before it does anything dangerous*.

```
int *p = nullptr;
int& i = *p;
foo(&i); // undefined
```

You might think that the `&` and the `*` cancel out and the result is as if you had written `foo(p)`, but the fact that you created a reference to a nonexistent object, even if you never carried through on it, invokes undefined behavior (§8.5.3(1)).

Related reading: What Every C Programmer Should Know About Undefined Behavior, Part 1, Part 2, Part 3.

Update: Broke the `&*` into two lines because it is the lone `*` that is the problem.

Raymond Chen

Follow