DOI:10.1145/3057284 http://cacm.acm.org/blogs/blog-cacm

Ending Null Pointer Crashes

Void safety, says Bertrand Meyer, relies on type declarations and static analysis.

Bertrand Meyer
Null-Pointer Crashes, No More
http://bit.ly/2i6w0nz
December 20, 2016

As an earlier article[5] emphasized, code matters; so do programming languages. While Eiffel is best known for its Design by Contract techniques, they are only part of a systematic design all focused on enabling developers to realize the best of their abilities—and eradicate from their code the sources of crashes and buggy behavior.

Talking about sources of crashes, one of the principal plagues of modern programs is null-pointer dereferencing. This term denotes what happens when you call *x.f*, meaning apply *f* (a field access or an operation) to the object that *x* references. If you want to define meaningful data structures, you need to allow "null," also known as Nil and Void, as one of the possible values for reference variables (for example, to terminate linked structures: the "next" field of the last list element must be null, to indicate there is no next element). But then you should make sure that *x.f* never gets called for null *x*, since there is in that case no object to which we can apply *f*.

The problem is particularly acute in object-oriented programming languages, where *x.f* is the major computational mechanism. Every single execution of this construct (how many billions of them occurred in running programs around the world since you started reading this?) faces that risk. Compilers for many languages catch other errors of a similar nature—particularly type errors, such as assigning the wrong kind of value to a variable—but they do nothing about prohibiting null pointer dereferencing.

This fundamental brittleness threatens the execution of most programs running today. Calling it a "billion-dollar mistake" as Tony Hoare did[1] is not an exaggeration. In his recent Ph.D. thesis[2], Alexander Kogtenkov surveyed the null-pointer-derefencing bugs in the Common Vulnerabilities and Exposures (CVE) database, the reference repository of information about Internet attacks. The resulting chart, showing the numbers per year, is edifying:

Beyond the numbers stand real examples, often hair-raising. The description of vulnerability CVE-2016-9113 (http://bit.ly/2mafdkJ) states:

There is a NULL pointer dereference in function imagetobmp of convertbmp.c:980 of OpenJPEG 2.1.2. image->comps[0].data is not assigned a value after initialization(NULL). Impact is Denial of Service.

Yes, that is for the JPEG standard. Try not think of it when you upload your latest pictures. Just for one month (November 2016), the CVE database contains null pointer vulnerabilities affecting products of the Gotha of the IT industry, from Google (http://bit.ly/2mfdAD2) and Microsoft (http://bit.ly/2muJImD) ("*theoretically everyone could crash a server with just a single specifically crafted packet*") to Red Hat (http://red.ht/2lXB5xS) and Cisco (http://bit.ly/2mMcueo). The entry for an NVIDIA example (at http://bit.ly/2lUREf8) explains:

For the NVIDIA Quadro, NVS, and GeForce products, NVIDIA Windows GPU Display Driver R340 before 342.00 and R375 before 375.63 contains a vulnerability in the kernel mode layer (nvlddmkm.sys) handler where a NULL pointer dereference caused by invalid user input may lead to denial of service or potential escalation of privileges.

We keep hearing complaints that "the Internet was not designed with security in mind." What if the problem had far less to do with the design (TCP/IP is brilliant) than with the languages that people use to write tools implementing these protocols?

In Eiffel, we decided that the situation was no longer tolerable. After the language had eradicated unsafe casts through the type system, memory management errors through garbage collection and data races through the SCOOP concurrency mechanism, null pointer dereferencing was the remaining dragon to slay. Today Eiffel is *void-safe*: a null pointer dereference can simply not happen. By accepting your program, the compiler guarantees that every single execution of every single *x.f* will find *x* attached to an actual object, rather than void.

How do we do this? I am not going to describe the void-safe mechanism in detail here, referring instead to the online documentation[6], with the warning it is still being improved. But I can give the basic ideas. The original article describing void safety (and giving credit to other languages for some of the original ideas) was a keynote at ECOOP in 2005[3]. Revisiting the solution some years later, I wrote[4]:

Devising, refining, and documenting the concept behind the mechanism presented here took a few weeks. The engineering took four years.

That was optimistic. Seven more years later, the "engineering" continues. It is not a matter of ensuring void safety; the mechanism was essentially sound from the beginning. The continued fine-tuning has to do with facilitating the programmer's task. Any mechanism that avoids bugs—another example is static typing—buys safety and reliability at a possible cost in expressiveness: you have to prohibit harmful schemes (otherwise you would not avoid any bugs), but you do not want to prohibit useful schemes or make them too awkward to express (otherwise it is very easy to remove bugs: just reject all programs!) or make them too awkward to express. The "engineering" consists of ever more sophisticated static analysis, through which the compiler can accept safe cases that simplistic rules would reject.

In practice, the difficulty of finetunign void safety mostly involve the *initialization* of objects. While the details of void safety can be elaborate, the essential idea is simple: the mechanism relies on *type declarations* and *static analysis*.

The void-safe type system introduces a distinction between "attached" and "detachable" types. If you declare a variable *p1* as just of type (for example) *PERSON* it can never be void: its value will always be a reference to an object of that type; *p1* is "attached." This is the default. If you want *p2* to accept a void value you will declare it as **detachable** *PERSON*. Simple compile-time consistency rules support this distinction: you can assign *p1* to *p2*, but not the other way around. They ensure an "attached" declaration is truthful: at runtime, *p1* will always be non-void. That is a formal guarantee from the compiler.

The static analysis produces more such guarantees, without particular actions from the programmers as long as the code is safe. For example, if you write

if *p2* /= **Void then** *p2.f* **end**

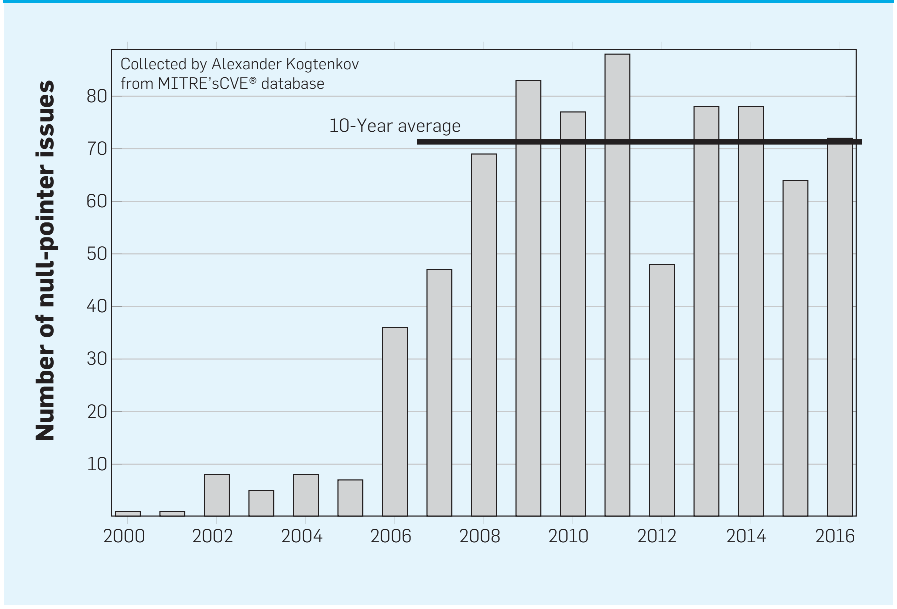
we know that things are OK. (Well, under certain conditions. In concurrent programming, for example, we must be sure that no other thread running in parallel can make *p2* void between the time we test it and the time we apply *f*. The rules take care of these conditions.)

The actual definition cannot, of course, say that "the compiler" will recognize safe cases and reject unsafe ones. We cannot just entrust the safety of our program to the inner workings of a tool (even open-source tools like the existing Eiffel compilers). Besides, there is more than just one compiler. Instead, the definition of void safety uses a set of clear and precise rules, known as Certified Attachment Patterns (CAPs), which compilers must implement. The preceding example is just one such CAP. A formal model backed by mechanized proofs (using the Isabelle/HOL proof tool) provides[2] solid evidence of the soundness of these rules, including the delicate parts about initialization.

Void safety has been here for several years, and no one who has used it wants to go back. (The conversion to voided safety of older, non-void-safe projects is not as painless.) Writing void-safe code quickly becomes second nature.

And what about your code: are you certain it can never produce a null-pointer dereference?

Null pointer issues (such as null pointer dereferencing) in Common Vulnerabilities and Exposures Database.



References

1. Hoare, C.A.R., Null References: The Billion-Dollar Mistake, August 25, 2009, http://bit.ly/2lAhgeP
2. Kogtenkov, A., Void Safety, ETH Zurich Ph.D. thesis, January 2017, http://se.inf.ethz.ch/people/kogtenkov/thesis.pdf.
3. Meyer, B., Attached Types and their Application to Three Open Problems of Object-Oriented Programming, in ECOOP 2005 (*Proceedings of European Conference on Object-Oriented Programming*, Edinburgh, 25-29 July 2005), ed. Andrew Black, Lecture Notes in Computer Science 3586, Springer, 2005, pages 1-32, http://bit.ly/2muJ8Ff
4. Meyer, B., Kogtenkov, A., and Stapf, E.: *Avoid a Void: The Eradication of Null Dereferencing*, in *Reflections on the Work of C.A.R. Hoare*, eds. C. B. Jones, A.W. Roscoe and K.R. Wood, Springer, 2010, pages 189-211, http://bit.ly/2lsNfN0
5. Meyer, B., Those Who Say Code Does Not Matter, *CACM*, April 15, 2014, http://bit.ly/1mNqout
6. Void safety documentation at eiffel.org: http://bit.ly/2lsS2xZ

Bertrand Meyer is a professor of software engineering at Politecno di Milano and Innopolis University.