DOI:10.1145/2209249.2209265

Article development led by acmqueue
queue.acm.org

Until our programming languages catch up, code will be full of horrors.

BY POUL-HENNING KAMP

My Compiler Does Not Understand Me

ONLY LATELY—and after a long wait—have a lot of smart people found audiences for making sound points about what and how we code. Various colleagues have been beating drums and heads together for ages trying to make certain that wise insights about programming stick to neurons. Articles on coding style in this and other publications have provided further examples of such advocacy.

As with many other educational efforts, examples that are used to make certain points are, for the most part, good examples: clear, illustrative, and easy to understand. Unfortunately, the flame kindled by an article read over the weekend often lasts only until Monday morning rolls around when real-world code appears on the screen with a bug report that just does not make sense—as in, "This can't even happen."

When I began writing the Varnish HTTP accelerator, one of my design decisions—and I think one of my best decisions—was to upgrade my OCD to CDO, the more severe variant, where you insist letters be sorted alphabetically. As an experiment, I pulled together a number of tricks and practices I had picked up over the years and turned them all up to 11 in the Varnish source code. One of these tricks has been called the red-haired stepchild of good software engineering and is widely shunned by most programmers for entirely wrong and outdated reasons. So let me try to legitimize it with an example.

Here is a surprisingly difficult programming problem: What do you do when `close(2)` fails?

Yes, `close(2)` does in fact return an error code, and most programmers ignore it, figuring that either: it cannot fail; or if it does, you are in trouble anyway, because obviously the kernel must be buggy. I do not think it is OK just to ignore it, since a program should always do something sensible with reported errors. Ignoring errors means you have to deduce what went wrong based on the debris it causes down the road, or worse, that some criminal will exploit your code later on. The one true ideal might appear to be, "Keep consistent and carry on," but in the real world of connected and interacting programs, you must make a careful determination as to whether it is better to abort the program right away or to soldier on through adversity, only to meet certain ruin later.

Realizing that "I have only a very small head and must live with it,"[1] sensible compromises must be made—for example, a trade-off between the probability of the failure and the effort of writing code to deal with it. There is also a real and valid concern about code readability—handling unlikely exceptions should not dominate the source code.

In Varnish the resulting compromise typically looks like this:

```
AN(vd);
AZ(close(vd->vsm _ fd));
```

`AN` is a macro that means Assert Nonzero and `AZ` means Assert Zero, and if the condition does not hold, the program core-dumps right then and there.

Yes, the red-haired stepchild I want to sell you is the good old assert, which I feel should be used a lot more in today's complicated programs. Where I judge the probability of failure is relevant, I use two other variants of those macros, `XXXAN` and `XXXAZ`, to signal, "This can actually happen, and if it happens too much, we should handle it better."

```
retval = strdup(of);
XXXAN(retval);
return (retval);
```

This distinction is also made in the dump message, which for `AZ()` is "Assert error" vs. `XXXAZ()`'s "Missing error-handling code."

Where I want to ignore a return value explicitly, I explicitly do so:

```
(void)close(fd);
```

Of course, I also use "naked" asserts to make sure there are no buffer overruns:

```
assert(size < sma->sz);
```

or to document important assumptions in the code:

```
assert(sizeof (unsigned short)
== 2);
```

But we are not done yet. One very typical issue in C programs is messed-up lifetime control of allocated memory, typically accessing a `struct` after it has been freed back to the memory pool.

Passing objects through `void*` pointers, as one is forced to do when simulating object-oriented programming in C, opens another can of worms. Figure 1 illustrates my brute-force approach to these problems.

In terms of numbers, 10% of the non-comment source lines in Varnish are protected with one of the asserts just shown, and that is not counting what gets instantiated via macros and inline functions.

A Method to the Madness

All this checking is theoretically redundant, particularly the cases where function A will check a pointer before calling function B with it, only to have function B check it again.

Though it may look like madness, there is reason for it: these asserts also document the assumptions of the code. Traditionally, that documentation appears in comments: "Must be called with a valid pointer to a foobar larger than 16 frobozz" and so on. The problem with comments is the

Figure 1. Mini objects.

```
        struct lru {
                unsigned                            magic;
                #define LRU_MAGIC                   0x3fec7bb0
                ...
        };
        ...
        struct lru *l;
        ALLOC_OBJ(l, LRU_MAGIC);
        XXXAN(l);
        ...
        FREE_OBJ(l);
```

The `ALLOC _ OBJ` and `FREE _ OBJ` macros ensure that the MAGIC field is set to the randomly chosen nonce when that piece of memory contains a `struct lru` and is set to zero when it does not.

In code that gets called with an `lru` pointer, another macro checks asserts the pointer points to what we think it does:

```
       int
       foo(struct lru *l)
       {
               CHECK_OBJ_NOTNULL(l, LRU_MAGIC);
               ...
```

If the pointer comes in as a `void *`, then a macro casts it to the desired type and asserts its validity:

```
       static void *
       vwp_main(void *priv)
       {
           struct vwp *vwp;
           CAST_OBJ_NOTNULL(vwp, priv, VWP_MAGIC);
           ...
```

Figure 2. Compile time asserts.

```
#define CTASSERT(x,z) _CTASSERT(x, __LINE__, z)
       #define _CTASSERT(x, y, z)     __CTASSERT(x, y, z)
       #define __CTASSERT(x, y, z)    \
               typedef char __ct_assert ## y ## __ ## z [(x) ? 1 : -1]
...
CTASSERT(sizeof(struct wfrtc_proto) == 32, \
Struct_wfrtc_proto_has_wrong_size);
```

compiler ignores them and does not complain when they disagree with the code; therefore, experienced programmers do not trust them either. Documenting assumptions so the compiler pays attention to them is a much better strategy. All this "pointless checking" grinds a certain kind of performance aficionado up the wall, and more than one has tried stripping Varnish of all this "fat."

If you try that using the standardized -DNDEBUG mechanism, Varnish does not work at all. If you do it a little bit smarter, then you will find no relevant difference and often not even a statistically significant difference in performance.

Asserts are much cheaper than they used to be for three reasons:

- Compilers have become a lot smarter, and their static analysis and optimization code will happily remove a very large fraction of my asserts, having concluded that they can never trigger. That is good, as it means I know how my code works.
- The next reason is the same, only the other way around: the asserts put constraints on the code, which the static analysis and optimizer can exploit to produce better code. That is particularly good, because that means my asserts actively help the compiler produce better code.
- Finally, the sad fact is that today's CPUs spend an awful lot of time waiting for stuff to come in from memory—and performing a check on data already in the cache in the meantime is free. I do not claim that asserts are totally free—if nothing else, they do waste a few nanojoules of electricity—but they are not nearly as expensive as most people assume, and they offer a very good bang-for-the-buck in program quality.

Intentional Programming

In the long term, you should not need to use asserts, at least not as much as I do in Varnish, because at the end of the day, they are just hacks used to paper over deficiencies in programming languages. The holy grail of programming is "intentional programming," where the programmer expresses his or her exact and complete intention, and the compiler understands it. Looking at today's programming languages, I still see plenty of time before progress goes too far and we are no longer stuck on compilers, but rather on languages.

Compilers today know things about your code that you probably never realize, because they apply a chess-grandmaster-like analysis to it. Programming languages, however, do not become better vehicles for expressing intent; quite the contrary, in fact.

It used to be that you picked a width for you integer variable from whatever register sizes your computer had: `char`, `short`, `int`, or `long`. But how could you choose between a short and a long if you did not know their actual sizes?

The answer is that you couldn't, so everybody made assumptions about the sizes, picked variable types, and hoped for the best. I do not know how this particular mistake happened. We would have been in much better shape if the fundamental types had been int8, int16, int32, and int64 from the start, because then programmers could state their intentions and leave the optimization to the compiler, rather than try to outguess the compiler.

Some languages—Ada, for example—have done it differently, by allowing range constraints as part of variable declarations:

```
Month : Integer range 1..12;
```

This could be a pretty smooth and easy upgrade to languages such as C and C++ and would provide much-needed constraints to modern compiler analysis. One particularly strong aspect of this format is that you can save space and speed without losing clarity:

```
Door_Height: Integer range
150..400;
```

This fits comfortably in eight bits, and the compiler can apply the required offset where needed, without the programmer even knowing about it.

Instead of such increased granularity of intention, however, 22-plus years of international standardization have yielded `<stdint.h>` with its `uint_least16_t`, to which `<inttypes.h>` contributes `PRIuLEAST16`, and on the other side `<limit.h>` with `UCHAR_MAX`, `UINT_MAX`, `ULONG_MAX`, but, inexplicably, `USHRT_MAX`, which confused even the person who wrote `od(1)` for The Open Group.

This approach has so many things wrong with it that I barely know where to start. If you feel like exploring it, try to find out how to portably `sprintf(3)` a `pid_t` right-aligned into an eight-character string.

The last time I looked, we had not even found a way to specify the exact layout of a protocol packet and the byte-endianess of its fields. But, hey, it is not like CPUs have instructions for byte swapping or that we ever use packed protocol fields anyway, is it?

Until programming languages catch up, you will find me putting horrors as those shown in Figure 2 in my source code, to try to make my compiler understand me. C

Related articles on queue.acm.org

Reveling in Constraints
Bruce Johnson
http://queue.acm.org/detail.cfm?id=1572457

Sir, Please Step Away from the ASR-33!
Poul-Henning Kamp
http://queue.acm.org/detail.cfm?id=1871406

Coding Smart: People vs. Tools
Donn M. Seeley
http://queue.acm.org/detail.cfm?id=945135

References

1. Dijkstra, E.W. Programming considered as a human activity (1965); http://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD117.html.

Poul-Henning Kamp (phk@FreeBSD.org) has programmed computers for 26 years and is the inspiration behind bikeshed.org. His software has been widely adopted as "under-the-hood" building blocks in both open source and commercial products. His most recent project is the Varnish HTTP accelerator, which is used to speed up large Web sites such as Facebook.