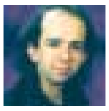


Kevlin Henney uses contracts to cut into the nuts-and-bolts world of C memory allocation

NO MEMORY FOR CONTRACTS

THERE IS A LATENT PERCEPTION THAT THE contract metaphor [1] for design applies only to object-oriented systems, in part because that is where the technique has been popularised [2]. However, it is useful wherever there is any separation between a caller and a called component – i.e. some kind of interface – and where the called component must be specified in some way to ensure clarity, stability and portability.

Taking a traditional function API as an example, the lifecycle of a heap-allocated object in C begins when it is allocated, using `malloc`, before then being properly initialised and used. As part of its use there may be cause to reallocate it, using `realloc`, which is more likely the case for an array than for a single object. Once its life is complete, the program performs any finalisation actions on the object as required and calls `free` to dispose of the raw memory. However, there is more to this picture than intuition would either suggest or accommodate. A contract-based perspective reveals a lot about both the benefits of employing the contract metaphor and the design shortcomings of the C memory management functions.

A contract-based perspective reveals a lot about both the benefits of employing the contract metaphor and the design shortcomings of the C memory management functions

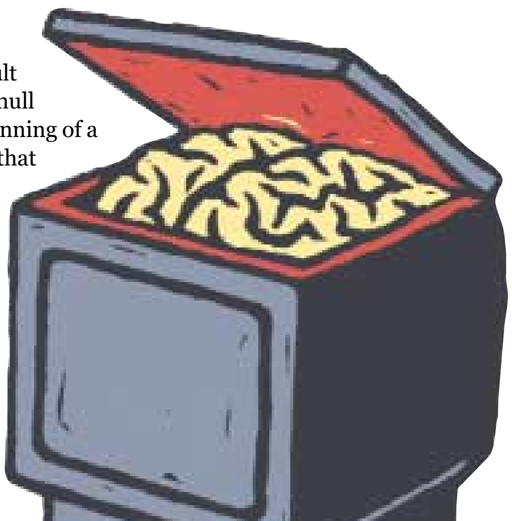
malloc

The signature for `malloc` is a fairly simple one:

```
void *malloc(size_t size);
```

Calling `malloc` equates to a request for `size` bytes to be returned as untyped memory. An intuitive contract would be as follows:

> *postcondition*: If allocation is successful the result returned is a non-null pointer to the beginning of a sequence of bytes that is suitably aligned for any type of object, dereferenceable in the first `size` bytes, and disjoint from any other dereferenceable object, otherwise a null pointer is returned.



This is both clear and precise. It is also not `malloc`'s contract. It is not so much that the actual standard specification of `malloc` is not written in contract form, although that is certainly true, but that it does not correctly reflect what happens when `size` is zero.

As it happens, the sample contract just shown works fine for the zero case: a successful allocation results in a non-null pointer from which zero bytes are dereferenceable. The only problem is that although reasonable, this is not what the C standard library is required to do. The contract in standard C [3] is a weaker, more mealy-mouthed contract that is most succinctly expressed in terms of two conditionally distinct postconditions:

> *postcondition where* `size > 0`: If allocation is successful the result returned is a non-null pointer to the beginning of a sequence of bytes that is suitably aligned for any type of object, dereferenceable in the first `size` bytes, and disjoint from any other dereferenceable object, otherwise a null pointer is returned.

> *postcondition where* `size == 0`: Either a null pointer or a non-null, non-dereferenceable pointer is returned.

In other words, if you call `malloc(0)` you may or may not get a null pointer if the call is successful. It is the "may or may not" that makes the contract a weak one. It is ambivalent where a contract should be unambiguous.

Although using `assert` on the result of `malloc` is considered poor practice in production code, because it tests a property of the code's environment rather than its essential correctness, in the controlled context of a unit test such assertions are acceptable, assuming that the environment has been instrumented to allow just correctness to be tested. So, with a little help from some auxiliary functions that we might include for the test harness, what might some tests against `malloc`'s contract look like?

```
// test successful allocation
ensure_sufficient_space(100);
result = malloc(100);
assert(result);
...
// test failed allocation
ensure_insufficient_space(100000);
result = malloc(100000);
assert(!result);
...
// test zero allocation
```

```
ensure_sufficient_space(0);
result = malloc(0);
assert(result || !result);
...
```

The last test is tautologous, so it doesn't look particularly convincing! However, you can't argue with it: it's what the contract says.

OK, so why worry? Asking for zero bytes seems an odd thing to do, so why waste time on an edge case? And what difference does it make what it returns anyway?

As we shall see, although asking for zero bytes is a boundary case it is not necessarily an edge case. Interfaces should be consistent [4], which the draft of the contract was but the second – and actual – contract is not. There are some knock-on consequences from this that are not immediately obvious, but we will come to those.

Another property of a well-considered interface is that it is well considered! It should be reasonable in the sense that it is grounded in reason [5]. What was the motivation behind choosing a weaker contract over a stronger and more concise one?

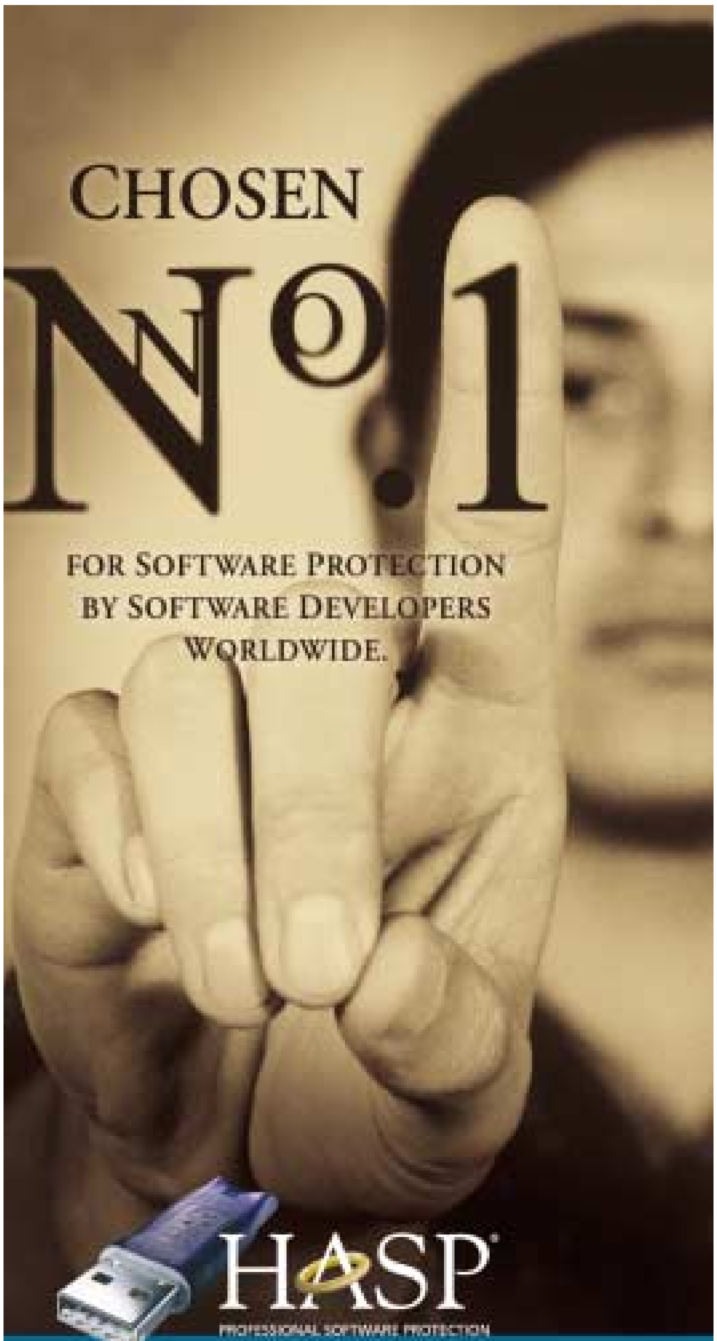
Zero-sized objects

The original ANSI C standardisation committee mistakenly assumed that a non-null result from `malloc(0)` was tantamount to accepting the existence of zero-sized physical objects – a philosophical conundrum best left unanswered in the scope of a programming language standard. They read `malloc(0)` returning non-null as "a successful request for a zero-sized object", rather than as "a successful request for a contiguous sequence of bytes with zero dereferenceable bytes", and understandably got cold feet. The second reading is subtly different, and sufficiently so that it bypasses the thorny existential issues. It also happens to be closer to what is going on: `malloc` does not work in terms of typed objects, and so it should not have been constrained by this channel of thinking.

Nevertheless, it is not just a matter of word play. C already has the concept of legal non-null pointers that cannot be dereferenced: pointers to one past the end of an array. Given an array `a` of `n` elements, the first dereferenceable element is `a[0]` and the last is `a[n - 1]`. The element past the end, `a[n]`, cannot be dereferenced and used except to have its address taken, i.e. `&a[n]` is OK. If one wishes to view this as a zero-sized object, fine, but that's not a particularly useful perspective.

The question of zero-length arrays has also been considered in other languages, such as C++ and Java, where a successful request for a zero-element array yields a non-null, empty array.

OK, we can now see from a number of angles that the reasoning used to legislate against zero-length allocations based on the implied existence of zero-sized objects was not sufficiently thorough or consistent. If this had been realised it would have saved an unnecessary and gratuitous change. `malloc` wasn't broken, so it shouldn't have been fixed. And that's the problem: the rationale for the C standard [3, 6] makes clear that `malloc`'s implied contract used to be the first draught that we presented (although on some older platforms `malloc(0)` was greeted by a segmentation fault). Instead of leaving it and clarifying any ambiguities across



Because of its unparalleled security, unsurpassed reliability, and ability to increase your software revenues, HASP4 is considered the world's best hardware-based software protection. No other company is more committed to the continual advancement of this proven technology for your success.

Call +44 (0)1753 622266 or email HASP.uk@eAladdin.co.uk for your **FREE HASP4 Developer's Kit today.**

Aladdin
SECURING THE GLOBAL VILLAGE
eAladdin.com

Aladdin Western Europe Ltd, Fairacres House, 2-3 Fairacres Industrial Estate, Dedworth Road, Windsor, Berkshire SL4 4LE

©2003 Aladdin Knowledge Systems Ltd. HASP is a registered trademark of Aladdin Knowledge Systems Ltd.

different versions, in line with the credo of standardising existing practice, the contract was instead diluted. Where the first contract states quite clearly that a non-null pointer would be returned, the second leaves the result to chance. It would even have been better if the logical consequence of the argument had been carried all the way through and null was categorically returned for zero-length allocations. Thus, the whole question of zero-sized objects was a complete red herring because the changed contract doesn't support or deny it! It's a fudge, not a fix.

calloc

And now for something not so completely different:

```
void *calloc(size_t array_count, size_t element_size);
```

Intended for array allocation, the contract for `calloc` function inherits `malloc`'s boundary case handling. Let's focus on the main part:

given: `size == array_count * element_size`.

postcondition where `size > 0`: If allocation is successful the result returned is a non-null pointer to the beginning of a sequence of zeroed bytes that is suitably aligned for any type of object, dereferenceable in the first `size` bytes, and disjoint from any other dereferenceable object, otherwise a null pointer is returned.

The operative word in there is "zeroed". Every bit of the allocated space is zero whereas with `malloc` the values are indeterminate.

At first sight it seems like a convenience worth having, but watch out: zeroing bits is not necessarily the same as nulling or `0.0`-ing. Although most platforms today represent null pointers and zero floating-point numbers with all bits set to zero, that is not always a portable assumption.

The effect of `calloc` is follows closely from its contract, i.e. the following

```
T *result = (T *) calloc(count, sizeof(T));
if(result)
{
    ...
}
else
{
    ...
}
```

is equivalent to the following

```
T *result = (T *) malloc(count * sizeof(T));
if(result)
{
    memset(result, count * sizeof(T), 0);
    ...
}
else
{
    ...
}
```

Although I don't generally recommend writing more code where less will do, the second version has the benefit of not misleading the reader into expecting more than is on offer: `memset` is fairly explicit in its lack of magic and so it less likely – but of course not impossible – that a reader will misconstrue byte zeroing as general value zeroing. And that assumes that zeroing is even the right thing: for most arrays of interesting data types you would probably want something more useful than zero as your starting state. The code also highlights that there is nothing particularly array oriented about `calloc`: replace a comma with an asterisk and you can use `malloc`.

So, the contract for `calloc` checks out, but its restricted applicability in the face of its apparent generality is its weakness. It is not really a first class memory management function and should not be considered as equal in rank to `malloc`, `realloc` or `free`.

free

The signature for free is easy to state:

```
void free(void *ptr);
```

Its functional contract is a little trickier to express:

precondition: `ptr` is either null or was returned by `malloc`, `calloc` or `realloc`, and has not since been deallocated by either `free` or `realloc`.

postcondition where `ptr != 0`: The space pointed to by `ptr` has been deallocated and cannot be dereferenced.

postcondition where `ptr == 0`: Nothing happened.

But testing it, on its own terms, is impossible. Working only within the scope of the API, it qualifies as an untestable proposition: without supplying auxiliary API functions, a test cannot be written to prove conformance (although you might find out soon enough in practice). Gödel's incompleteness theorem is not just the idle plaything of logicians; it is quite tangible in the world of APIs and testing!

Irony

There is a certain irony that in allowing `free` to accommodate null the original justification [3, 6] was given as being "to reduce the need for special-case coding" and yet breaking `malloc` introduced the need for special-case coding! However, the idea that deallocating nothing causes nothing to happen is also a reasonable one for `free`, regardless of whether special-case coding is taken into account

realloc

And so to reallocation:

```
void *realloc(void *ptr, size_t new_size);
```

I will leave the exercise of actually writing `realloc`'s contract for the reader, so let's look at what needs to go into it. Intuitively, as suggested by its name, `realloc` takes a pointer to already allocated memory and tries to reallocate it to a new size. This may involve a new allocation, copying from the old space and then deallocation of the old space, or it may discover that the underlying block of memory it is pointing to is already sufficiently large to accommodate the new requested size, returning the original pointer without any reallocation or copying. Intuitively that would be reasonable. However, in real life `realloc` has ambitions beyond our intuition. It tries to do and be nearly everything through its narrow interface:

- It can be an allocator of new memory: if `ptr` is null it will behave like `malloc(new_size)`.
- It can be a deallocator of allocated memory: if `new_size` is zero it will behave like `free(ptr)`.
- It can be a reallocator of allocated memory: if `ptr` is non-null and `new_size` is non-zero it will attempt to do what you would reasonably expect a function named `realloc` to do!

`realloc` has been rightly criticised for being a "one-function memory manager" [7]. It is a classic example of a function with weak cohesion. It turns out that much of its uncohesiveness can be blamed on `malloc`, whose weak postcondition is viral.

Empty arrays

A not uncommon scenario is to use an array for holding some kind of data – whether a data set or a lookup table – and for that array to start off empty and then be resized as it grows. This means kicking off with `malloc(0)` and then successively calling `realloc`. Ah, but what is the result of `malloc(0)`? And therein lies the problem. To resize a previously empty sequence of bytes requires that null is a permissible argument to `realloc`, because null is a permissible success result for `malloc(0)`. In this case `realloc` plays the same role as a plain `malloc`. The additional emulation of `free` when called with a zero size argument is gratuitous, although at least it is the one place that is consistent with the zero-sized-objects-are-not-permissible line of reasoning!

The significance of the invention of zero should not be underestimated [8], but there is a lot of misinterpretation and mysticism associated with it. In this particular case, it has led to a weak contract in one function that has, in turn, undermined the cohesion of another function. ■

References

1. Kevlin Henney, "Sorted", *Application Development Advisor*, July 2003
2. Bertrand Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997
3. *The C Standard*, Wiley, 2003
4. Kevlin Henney, "Form Follows Function", *Application Development Advisor*, March 2004
5. Kevlin Henney, "Conventional and Reasonable", *Application Development Advisor*, May 2004
6. *The ANSI C Rationale*, www.lysator.liu.se/c/rat/title.html
7. Steve Maguire, *Writing Solid Code*, Microsoft Press, 1993
8. Robert Kaplan, *The Nothing that Is*, Penguin Books, 2000

Kevlin Henney is an independent software development consultant and trainer. He can be reached at www.curbralan.com.

VERSANT

Extremely smart:

The Proof is in the Code.

Are your development cycles getting shorter? Is your requirements list getting longer? How do you find the time to innovate and be creative without jeopardizing the next milestone?

If your database component is forcing you to write large amounts of mapping code, you should consider an alternative. Versant's database technology offers Transparent Persistence – Eliminate all that error prone mapping code. You never have to debug an object-relational mapping layer again! Instead you could reduce your time-to-market, invest in creativity, develop unique functionality in your system and gain a competitive advantage. Work smart – Extremely Smart!

Transparent Persistence for Java, C++ and .NET

↳ www.versant.net