Some Were Meant for C

The Endurance of an Unmanageable Language

Stephen Kell
Computer Laboratory
University of Cambridge
Cambridge, United Kingdom
stephen.kell@cl.cam.ac.uk

Abstract

The C language leads a double life: as an application programming language of yesteryear, perpetuated by circumstance, and as a systems programming language which remains a weapon of choice decades after its creation. This essay is a C programmer's reaction to the call to abandon ship. It questions several properties commonly held to define the experience of using C; these include unsafety, undefined behaviour, and the motivation of performance. It argues all these are in fact inessential; rather, it traces C's ultimate strength to a *communicative* design which does not fit easily within the usual conception of "a programming language", but can be seen as a counterpoint to so-called "managed languages". This communicativity is what facilitates the essential aspect of system-building: creating parts which interact with other, remote parts—being "alongside" not "within".

CCS Concepts • **Software and its engineering** → **General programming languages**; *Compilers*; • **Social and professional topics** → *History of programming languages*;

Keywords systems programming, virtual machine, managed languages, safety, undefined behavior

ACM Reference Format:
Stephen Kell. 2017. Some Were Meant for C. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Vancouver, Canada, October 25–27, 2017 (Onward!'17),* 18 pages.
https://doi.org/10.1145/3133850.3133867

Onward!'17, October 25–27, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5530-8/17/10…$15.00
https://doi.org/10.1145/3133850.3133867

1 Introduction

> While some were meant for sea, in tug-boats
> 'Round the shore's knee,
> (Milling with the sand,
> and always coming back to land),
> For others, up above
> Is all they care to think of,
> Up there with the birds and clouds, and
> Words don't follow.
>
> —Tiny Ruins, from "Priest with Balloons"

I am not ashamed to say that I program in C, and that I enjoy it. This puts me at odds with much of programming language discourse, among both researchers and influential practitioners, which holds that C is evil and must be destroyed. If only we had a "safe systems programming language"! If only we could eke out a little more performance in implementations of other languages, to remove the last remaining motivation for using C! If only we could make "the industry" see the error of its ways! Then C would be eradicated, and there would be much rejoicing.

I am a "systems programmer". It doesn't mean I hack kernels, so much as that I build systems—pieces of infrastructure that integrate multiple interacting parts, and sit underneath application code. Programming in C feels *right* for doing this; it has a viscerally distinctive feeling compared to other, safer, higher-level languages. Certainly, today's experience of programming in C remains, despite certain advances, unforgiving. But I have never felt C to be an encumbrance. C is not a language I use because I'm stuck with it; I use it for positive reasons. This essay explores those reasons and their apparent contrast with conventional wisdom.

2 Two viewpoints

The lyric from which this essay borrows its title evokes two contrasting ways of being: that of the idealist who longs to be among the clouds, and that of the sea-farers who carry on their business on the planet's all-too-limiting surface. The idealist in the song is a priest, who takes literally to the clouds: one day, clutching at helium balloons, he steps off a cliff edge, floats up and away, and

is never seen again.[1] Meanwhile, the tug-boats far below symbolise another way to live: plying their trade along the rocky shoreline that is nature's unmovable constraint. The seafarers' perspective is limited and earthbound, shaped and constrained by hard practicality.

Both viewpoints are familiar to anyone interested in programming. The singer sympathises with the priest, as can we all: it is natural to dream of a better world (or language, or system) overcoming present earthly constraints, moving over and beyond the ugly realities on the ground. But the priest's fate is not a happy one. Meanwhile, just as the tug-boat crews are doing the world's work, the C language continues to be a medium for much of the world's working software—to the continued regret of many researchers.

As researchers in language design and related topics, we face a recurring question: how much ought we to idealise the rocks and waves inhabited by practitioners? We can idealise not only *how* gadgets work (machines, software), but also *why* particular trade-offs win out in particular cases—including why people use C. There is much valuable work to be done in idealised terms, on designs, or in clean-slate fashion, or atop theoretical rather than practical substrates. But the balloonist's view does not seem to explain the endurance of C, given its attendant problems; the sea-farers seem to live in some other, incommensurable reality. To understand this phenomenon, we will have to think differently. Over the remainder of the essay, I'll argue the following points (each in its own section), sharing the theme that we need to think less about languages as discrete abstractions, less hierarchically in general, and more about the *systems* which embody languages—seeing *language implementations* as parts of those systems, and not shying away from contextual details associated with implementation concerns and non-portability. Despite this prevailing preference for talking and thinking about languages in a discrete sense, I will also note certain ways in which we have the habit of confusing C's implementations with the language itself.

- At application level, C's continued popularity owes mostly not to performance but to a host of unglamorous yet technically hard problems in the general areas of migration, interoperation and integration. These are widely known but continue to be neglected and undervalued as topics of research per se—perhaps, I suggest, because they are about system design and engineering research (no contradiction) more than they are about languages (§4).
- For true "systems" programming (as I will define), C's benefits are of another kind. Again, performance is not the issue; I will argue that *communication* is what defines system-building, and that C's design, particularly its use of *memory* and explicit *representations*, embodies a "first-class" approach to communication which is lacking in existing "safe" languages (§5).
- I consider the valuable but limited paradigm of *managed* languages, and argue that it is not the only way to achieve *safe* languages; in fact managedness rules out first-class communication, so *cannot* replace C for system-building. Managed approaches are founded on hierarchical abstraction; I sketch an alternative "mediated" approach which is relatively heterarchical, suits alternative styles of program-level reasoning, but appears just as capable of safety guarantees, albeit at the level of a system and not a language (§6).
- Issues of unsafety and undefined behaviour in C are widely misunderstood, particularly in the extent to which undefined behaviour is motivated by performance, and also in what pertains to implementations of C versus to the language specification. I elaborate on this, helped by some choice words from a well-known system-builder and C programmer (§7).

I'll begin by recapping some conventional research viewpoints on C that form the basis of dispute.

3 Pieces of a debate

Some months back, social media conspired to point me towards Trevor Jim's article "C doesn't cause buffer overflows; programmers cause buffer overflows" [Jim 2015]. The title is ironic; the article is one of a loose series, best summarised as "don't use C [or] C++; use safe languages" with a tone not far short of outrage. C causes huge harm; why so little impetus towards change?

The advice to prefer safe languages is hard to argue with as far as it goes. The author's credentials as a co-designer of Cyclone [Jim et al. 2002], a safe language designed as a replacement for C, are no coincidence. Indeed, it is fair to be outraged. Untrapped buffer overflows are a ridiculous way for software to fail in the 21st century. The list continues to grow; 2014's Heartbleed [CERT 2014] is a high-profile new tip of an ancient iceberg.

Could the solution be as simple as refraining from using languages that allow these errors? I contend not; at least, before we can adopt this plan, we need to answer some questions. To what extent are existing safe languages actually capable of expressing the things that "unsafe" languages are used for? Is it a *language* that is

[1] This is, more-or-less, the true story of Father Antonio Adelir de Carli, whose ill-fated April 2008 balloon flight from the Brazilian city of Paranagua was a fundraising effort towards a "spiritual rest stop" for lorry drivers.

"safe", or an implementation of one? Rather than throwing away all that code, is it possible simply to implement C safely, and would the result be useful?

The article was circulated on social media in a way which reached both myself and some genuinely distinguished computer scientists. The resulting conversation was enlightening—not so much for technical insight, but rather for its recirculation of popular belief and folklore. I was alarmed at what I read, because world experts in programming languages were taking for granted a "fact" I intuitively felt sure to be false: that the only technical issue keeping people using C is performance.[2]

> "I find it hard to believe that there are many situations where the extra efficiency (often imagined) is really merited when you could have written the same code more safely in Go or some other language that provides more support to avoid memory management errors."

A familiar claim was that better languages are the solution, if only they can be made fast enough.

> "With Rust, you can have it both ways. … I agree that often one does not need the extra efficiency, but for certain platforms and applications it seems to be crucial (or else why wouldn't people have abandoned C a long time ago)."

Why indeed. Others instead believed industry to be malignly opposed to improvements that are readily available.

> "The very idea that C prevailed over the various Pascal-like languages available in the 1970s – and that practices were not reformed after the Morris worm in 1988 – is proof that the industry thought it could get away with serving up crap to its customers, protected by warranty disclaimers. They kept doing it for decades, until Windows practically drowned in a tide of malware. Shame."

> "Languages like C and C++ are popular because they make it harder to write good code, so improve wages and job security of programmers."

These beliefs are convenient, but seemed implausible to me. Thankfully I was not completely alone in finding C an enabling language; one respondent ventured as follows.

> "I enjoy C and writing C but I do it in a well-managed, safe space for my enjoyment, and certainly not out in public where I might hurt somebody."

I am usually the first to agree that the discipline of software is infuriatingly slow to advance. We, as researchers, are all implicated in this. No doubt industry is guided partly by groupthink and inertia—but also in large part by hard economic incentives. Rather than simple conservatism or a bad attitude within industry, it seems more plausible that our research is not doing the complete job. Unless we can understand the real reasons programmers continue to use C, we risk researchers continuing to solve a set of problems that is incomplete and/or irrelevant, while practitioners continue to use flawed tools.

If these were not the true reasons, then what are? A couple of other comments gave some insight. The first was about tools.

> "I honestly can't imagine writing C++ without ASAN anymore. Many memory bugs are just too hard to solve quickly without tools."

The second was about availability, with an intriguing hint about interoperability (or lack thereof) between numerical domains and the domain of the machine.

> "When I first studied programming, C and Fortran were the essential languages to know. Engineers worked with Fortran, and embedded platforms came with compilers only for C."

I decided to write down my own list of reasons why I believed people really use C. My first few attempts mostly re-treaded a familiar argument affecting mostly application code; I will recount these in the next section. I later realised that systems code is different; I will return to this in the section following (§5).

4 Application code: bad news, bad news

Many of the issues keeping application code in C are known—but were not remarked on by the respondents above, and are nothing to do with performance. Rather, they have to do with integration. Somehow, integration issues do not come to mind as easily; "faster safe languages" is seen as the Important Research Problem, not better integration.

[2] Direct quotations are used with permission of their authors—who remain anonymous, since their identities would only be a distraction here. In two cases where I did not hear back from the authors before the publishing deadline, I have instead used paraphrases of the original quotations.

What do I mean by "application code"? Any definition is at best a rule of thumb, but we can usually recognise application code when we see it. Programs and libraries alike are application code if they are application-directed—for example, libjpeg concerns pictures, and libssl concerns sending messages privately and securely. By contrast, large parts of of operating system kernels, language runtimes, or infrastructure libraries like libc, concern the realisation of the generic computational abstractions on offer (instruction sets, processes, storage devices etc.). These units of code lack particular human-facing application, but exist as vital cogs in the overall machine, and gain function from their mutual meshing-together. (As I will continue to elaborate later, this interacting, communicating essence is particularly suited to coding in C.)

4.1 Better is worse

Decades earlier, considering the state of Lisp implementations, Richard Gabriel [1994] mused that C came from a different design school than Lisp and other more academic-flavoured projects. (The essence of C's "worse is better" is the preferred trade-off among simplicity, correctness, consistency and completeness: it values simplicity the most, and tolerates minor incorrectness, inconsistency and incompleteness. In contrast, Lisp's "the right thing" school prefers to avoid these, and pay extra in complexity.) Almost orthogonally, though, he also observed that something about C's "worse is better" design presented far fewer integration difficulties to the application programmer, relative to what had yet been achieved with Lisp under the "right thing".

> "In the worse-is-better world, integration is linking your .o files together, freely intercalling functions, and using the same basic data representations. You don't have a foreign loader, you don't coerce types across function-call boundaries, you don't make one language dominant, and you don't make the woes of your implementation technology impact the entire system."

My own list began largely (I later realised) as a paraphrasing and elaboration of Gabriel's points, also sharing some similarity with remarks by Philip Wadler [1998] some years later. Since it has a specific focus on C code today, it bears recapping.

Language migration: all-at-once or not-at-all Like any language, C persists partly because replacing code is costly. But perversely, the implementation technologies favoured by more modern languages offer especially unfavourable effort/reward curves for migration. Migration all at once is rarely economical; function-by-function is probably the desired granularity. Essentially all high-level languages' virtual machines offer foreign function interfacing (FFI) systems (consider Java's JNI, Haskell 98's standard FFI, and so on), which make this near-impossible. Gabriel refers to "[coercing] types across function call boundaries" and to differing data representations. Although simple libraries exchanging only primitives can get by with present FFIs (consider for example the popular GNU multi-precision arithmetic library gmp, used from many languages), functions exchanging non-trivial data structures hit a wall: these structures must be shared among code in multiple functions, hence (if we are migrating function-by-function) between multiple languages. So in which language are they defined? Either they must be implemented twice, and deep copying-and-coerce code inserted, or access must be functionalised into simple primitive-signature operations and those functions themselves FFI-wrapped. This forces a primitive interface onto code which (by definition) does not suit it.

Transitive economies Many libraries are written in C. Libraries lie *under* their clients so are more likely to be talking directly to the hardware or operating system (usually easier in C than in other languages) or, transitively, closer to code that does this. The performance argument is relevant here, albeit non-obviously. Since libraries "sit under" a fan-in of many clients, they are more likely to justify the investment of using less productive languages in return for faster or leaner results. The client programmer *ought* still to have a free choice of languages, but C "leaks" across the join, because the pain of language boundaries presents strong incentives to using C in the client too.

Primitive tools, great works New materials and tools, whatever their merits, would hardly justify rebuilding the pyramids of Egypt. Why rewrite code at all? Primitive tools can still produce great work. The proposition of rewriting codebases is an expensive one, and an *increase* in bugs is highly likely in the short term. If (as I argue in §6) it is feasible to implement C in a dynamically safe manner, the existing code's apparent security and debuggability drawbacks are obviated too.

Linking versus dominance Gabriel notes the benefit of linking .o files together and avoiding the dominance of one language over another. This symmetric, flat, language-agnostic "linking" composition operator is the complete opposite of present foreign function interfaces' offerings. These provide only directional, hierarchical notions of "extending" and (often separately) "embedding" APIs [Ierusalimschy et al. 2011]. The former lets one introduce foreign code (usually C) as a new primitive in the VM, but only if the C is coded to some VM-specified interface.

The latter lets foreign code call into VM-hosted code, but again, only using an API that the VM defines. "A C API is enough" is the VM engineer's mantra. The resulting glue code is not only a mess, but worse, is required to be in C… all of this for a programmer trying *not* to use C! This is no ordinary C code, either: the programmer faces huge amounts of VM-specific detail, often implementation-specific. Consequently, the worse-yet-better approach of sticking with C everywhere often wins. Although other outcomes sometimes prevail, whether grasping the FFI nettle or rewriting a whole codebase in the new language, the effort/reward curve should not be this punishing.

(Higher-level FFI approaches based on code generation attempt to soften the curve, but with limited success to date. A thorough critical review of these approaches could fill another essay; to this author's knowledge, all stop short of eliminating the "dominance" to which Gabriel refers—either by demanding buy-in to a single virtual machine, by non-idiomatically embedding one language in another, by requiring repetition of "foreign" interface definitions in an ad-hoc notation, by failing to accommodate a sufficiently large majority of existing C code, and/or by coming with severe compiler- or VM-specific limitations.)

The "languaginess" of interfaces A module's implementation language ought to be an implementation detail. Current infrastructure makes it an essential part of its interface; APIs exist "in" a language, not apart from them. This relates both to the glue coding we just discussed, and the "foreign loader" mentioned by Gabriel. Our best answers so far are to use interface definition languages (IDLs) of systems such as COM and CORBA, the infamous distributed computing middlewares. Aside from their specialisation towards over-the-wire marshalling, hence their relative unsuitedness to in-memory interoperation, there are clear downsides. The prescriptive code-generation paradigm forces a peculiar and usually unwanted style of coding (including peculiar data types) on the programmer. Unless one has "bought in" to a particular IDL from the outset, there is usually no economical path for retroactively migrating to the IDL later [Kaplan et al. 1998].

Interface co-evolution Compounding the cost of interfaces' languaginess is that interfaces change, and most co-evolve with the implementations they attempt (but succeed only imperfectly) to encapsulate. Programming most often proceeds by targetting the interfaces of code written previously. We seldom question this, but it is a strange property. A hardware designer, by contrast, does not design an integrated circuit (IC) by considering the fine details of some specific other IC that is already on the market. Although "abstraction layer" software interfaces *can* be devised separately from their implementation, this is not the default; it requires additional up-front investment, and works only to the extent that future changes can be anticipated. FFIs, IDLs and other techniques based on manually gluing user code onto auto-generated ("stub"-style) interfaces, amplify the cost of this direct dependency by spreading interface details further: not only between the modules on either side, but into a third (usually hairy) glue module.

4.2 Languages, tools, systems

A related issue inhibiting moves away from C is that newer languages offer immature tools—including being (paradoxically) the languages least likely to come with tool support for effectively mixing languages. We conceive new languages as new worlds; this is the balloonist's way.

Even Google, which has less reason than most to be intimidated by creating a fresh world, has struggled with this, notably in the support for interactive debugging in the Go language. This remains in purgatory, at least in its ground-up `golang` implementation.[3]

Contrast this with `gccgo`, the tug-boat worker's approach to the same language design. It is an alternative implementation which integrates into a wider, richer, more complex, quirkier and more limiting programming system: the GNU `gcc-binutils-gdb` triumvirate. This infrastructure does not decompose neatly along language lines; it is not a collection of cleanly delineated language implementations, each with tools. Rather than adding discrete tools on the side, the system has a common basic fabric. Cognate perspectives can be found elsewhere: at SPLASH 2012, Jim Coplien's infamous keynote expressed a scepticism about tools, which David Ungar echoed with the analogy that "if every bolt on my car had its own little handle, I wouldn't need to go and get a wrench". It is this call to emphasise *systems* that I echo here—and as I will note later (§5.4), it accords strongly with the customs of C implementation, if not the C language per se.

The moment we start talking about FFIs and the like, we are talking about language implementations, not language designs. Most FFIs are unashamedly implementation-specific. Although a few (such as Java's) are standardised per-language, these tend to be the least usable, since the desire to accommodate diverse implementations leads to extra indirections and only weak guarantees. Being "standard" also does not stop alternatives from springing up—the last 15 years have seen

[3] See Rob Pike's post "The state of `gdb` support" to `golang-dev` on 2014/3/5, since when the status within that project has not changed significantly to my knowledge. The technical issues here are subtler and more complex than I have properly explored, and third-party debuggers do exist, but the omission remains striking.

at least CNI [Bothner 2003], JNA[4], GNFI [Grimmer et al. 2013], Panama[5] and no doubt others.

If better solutions are there to be found at the implementation level, we will only find them if the corresponding *problems* are viewed as sufficiently research-worthy. Gabriel believed the problems to be matter of "implementation technology"—not inherent to the Lisp approach.

> "The very best Lisp foreign functionality is simply a joke when faced with the above reality. Every item on the list can be addressed in a Lisp implementation. This is just not the way Lisp implementations have been done in the *right thing* world."

Twenty-six years later, most high-level language implementations struggle with essentially the same problems. Perhaps the problems are intractable after all; or perhaps, simply, cultural inertia is powerful, and energy has yet to be spent on deeper re-thinking. The "systems" versus "languages" dichotomy, also expounded by Gabriel [2012], fits well here. Focus on each language as a discrete, all-surrounding "world" neglects the fact that languages *need not* be the bearers of all innovation, and that *systems*, with language implementations as one part, make up the fabric of any programming environment. Balloonists continue to do what looks to be "the right thing", creating new worlds. Down around the shore, these new worlds remains out of reach.

5 Communicative code

I have talked about application code; what about systems code? I will return shortly to what exactly that means; first, let's look at some code that definitely fits this description.

5.1 Objects in space

One famous piece of operating system code concerns a trick for loop unrolling, later to be known as "Duff's device". (The clearest complete explanation is probably that by Simon Tatham [2000].) The trick concerns how to unroll a simple loop of the following form.

```
do { /* unsigned count > 0 assumed */
    *to = *from++;
} while(--count > 0);
```

Leaving aside the cleverness of Duff's unrolling solution (not shown, and not relevant to us here), the loop at first appears useless, because all but the last write to *to is never read.[6] The code is nevertheless useful, because *to does not refer to a program variable; in fact it refers to a memory-mapped register on which all the writes will have some effect.[7] Despite that, the language lets us access it just like any other object in memory.

More generally, C's notion of *memory*, arranged in an address space, allows code to *address* (point to) and *access* (read, write, call) objects inhabiting that space. Unlike most other languages, those objects need not have been defined within our program. In fact they even need not behave in the same way as such objects. Despite this, in all cases we access them in the same, uniform way.

Besides our own program's objects in memory, there may be "alien" objects with the same memory-like behaviour, such as memory populated by the operating system. There may also be objects that do not even behave like memory, such as device registers. In both cases, these objects exist to facilitate communication with the world *outside* the program. In C, these objects are "first-class": with them we can perform all the usual operations that we can perform on our own program variables. We can read them and write them; take their address and pass it around; or (if appropriate) call them.

When programming in C, my own mental model views memory as a space of communication channels. The C abstract machine's notions of reads and writes are access to channel endpoints, roughly in the sense of Shannon and Weaver [1949]. Channels transmit symbols, drawn from an alphabet; most channels have memory-like "storage" semantics, meaning all but the last-transmitted symbol are lost. But other channels, like the register in Duff's code, may have other semantics more akin to those of typical communication channels.

C's explicit access to *representations*, in the form of raw `chars`, also fits this channel model: this is what lets us *compute over symbols*, even symbols that are not native to the language implementation. Crucially, this facility is bidirectional. We can see any program variable as `chars`, and also interpret any `chars` as some program type. This lets us exploit whatever commonality does exist between these alphabets, avoiding unnecessary "byte-level" programming. For example, code written to run on a big-endian machine can access the integer fields in a network packet header directly, even in a buffer that is supplied by the operating system rather than defined in C code.

Here is another snippet, showing an example of an alien object that is not a register but plain old memory. Like later snippets in this section, it comes from code in

[4]Java Native Access, documented at https://github.com/java-native-access/jna as retrieved on 2017/4/21.
[5]documented at http://openjdk.java.net/projects/panama/ as retrieved in 2017/4/21.

[6]Given the absence of any kind of synchronisation, we can also assume that it is not intended to be observed by concurrently executing code.
[7]In modern C, *to would be declared `volatile` to signify this, preventing a sufficiently clever compiler from eliding any writes.

one of my own projects. It is using shamelessly platform-specific knowledge of the process's memory layout to access some data supplied by the operating system in a structure called the *auxiliary vector*, provided by all modern Unix systems (since AT&T's System V r4).[8]

```
for (const char **p_str = &environ[0]; *p_str; ++p_str)
{
        if (*p_str > (const char*) stackptr
                && *p_str < (const char *) stack_upper_bound)
        {
                /* It's pointing into the auxv's environ block. */
                uintptr_t search_addr = (uintptr_t) *p_str;
                search_addr &= ~(ALIGNOF(ElfW(auxv_t)) - 1);
                ElfW(auxv_t) *searchp = (ElfW(auxv_t) *) search_addr;
                while (!( /* complex cond */ ))
                {
                        searchp = (ElfW(auxv_t) *) ((uintptr_t) searchp -
                                ALIGNOF(ElfW(auxv_t)));
                }
                ElfW(auxv_t) *at_null = searchp;
                assert(at_null->a_type == AT_NULL
                        && !at_null->a_un.a_val);
                /* ... break; */
        }
}
return NULL;
```

Unlike the use of Duff's device, we are clearly talking to memory—but memory that came from the operating system, not from anything defined within our C program or its libraries. First, we walk the environ array, which contains a mix of pointers to "native" strings (defined by the program) and "alien" strings created by the operating system (OS). The fact that we can create such a mixed structure at all is itself an instance of the first-classness afforded to alien objects. Next, we use environment-specific knowledge to identify a string that came from the OS, and to parse the memory lying nearby it. This memory is in an OS-defined structure, called the auxiliary vector, that lives at the top of the initial stack (in processes on an ELF[9]-based System V-style Unix environment). This structure contains various other data passed to us by the operating system.

This is obviously environment-dependent code. It feels decidedly "unsafe". (Indeed, at present, it is. I will keep the issue of safety somewhat sidelined to begin with, in favour of focusing on expressiveness—but I will return to it in §6.) Nevertheless, it is possible to enumerate the assumptions under which the code is correct. These assumptions involve memory: that the address space is laid out a certain way, that the auxiliary vector's representation obeys a certain format, and that certain properties of environment strings' locations may be relied on. These properties hold in the System V-style Unix process environments that the code targets.

A fairly conventional C bounds checker could do a pretty good job on this code, if only it had information about the bounds and internal structure of the auxiliary vector. Tool support conceived "for C", however, cannot provide this information, because the auxiliary vector is an alien object: it does not come from code written in C. Systems code often accesses "alien" structures, defined by the language implementation, the linker, the operating system, and so on. Many of these implementation-specific details are routinely documented and specified in application binary interface (ABI) specifications such as that of the Santa Cruz Operation [1997]. These supplement the C language with a large number of properties, including about data representation and address space layout, against which systems programmers may justifiably rely when targeting systems conforming to that ABI. (In this case, not quite all of the relied-on properties are documented, so arguably the code is not portably correct. However, the wider point stands: properties originating outside a language, and outside any code written in that language, may be used to reason about code in that language.)

5.2 Alien memory

Our next snippet is doing instrumentation of some target code, in the form of machine instructions. It does so by iterating over all instructions in a range of memory, and invoking a callback on each.

```
static void walk_instrs(unsigned char *pos, unsigned char *end,
    void (*cb)(unsigned char *, unsigned, void *), void *arg)
{
    unsigned char *cur = pos;
    while (cur < end)
    {
        unsigned len = instr_len(cur, end);
        cb(cur, len, arg);
        cur += (len ? len : 1);
    }
}
```

This tiny piece of code is doing something that would be completely inexpressible in any current safe language I know. The code doesn't care where the instructions came from. It can read and write instructions appearing in any executable region of memory. It can include virtual machines' JIT-compiled regions, code compiled from languages other than C, and so on. Our C implementation can operate on data (in this case code, but code is data) that it knows nothing about, from a region

[8] It is rather an accident that this code is necessary—it happens that no portable API exists for getting access to this particular structure, although some systems do provide a getauxval().

[9] Executable and Linkable Format—the principal binary file format on modern Unix platforms, originally defined in System V r4 [AT&T 1990].

of the address space that it has no role in managing or creating.

Contrast this with a safe language. How would we even begin? To iterate over something using the natural features of the language (object references, array indexing), it would need to be an array *managed* by that same language implementation. Alternatively, at best there might be an API exposing access to peek and poke all of memory—but using that would mean we are no longer using the basic language features, but a tiny embedded mini-language (the API) which is unlikely to be as expressive or convenient. Put differently, in such a scenario, communication is no longer first-class. (This is not remedied by operator overloading or other niceties, which would at best paper over this distinction.)

Actually, I lied a little: the code does care where the instructions came from. It cares so that it can exclude *itself* from instrumentation. To do so, its caller (not shown) refers to some global variables marking the beginning and end of its own text segment. Interestingly, these variables are not defined in any C code at all. Rather, they are defined by linker inputs—an adjoining part of the toolchain. As I (and co-authors) argued recently [Kell et al. 2016], linker inputs can be thought of as a distinct programming language in their own right. As I cover shortly (§5.4), interoperability with "linker-speak" from C is assured—not by the language, not by an FFI, but by the implementation norms of C, as a cooperating piece in an open-ended toolchain. Attempts to re-implement or replace C neglect this at their peril.

5.3 Systems as interactions

It is tempting to think that the code in these snippets is a special case. It is clearly environment-dependent. The second snippet contains meta-level, reflective code, not base-level application logic. And it clearly occurs deep within some infrastructure. Is this really contributing to the argument about C's utility?

I'd happily agree that the most compelling applications of C are, indeed, something to do with this kind of code. I call it "systems code"—not necessarily referring to kernel code or the like, but about code whose job is *interaction* or (equivalently) *communication*. A system, by definition, consists of multiple interacting parts.

This is communication in a stronger sense than that in which a client communicates with a library. Whenever we program against an interface, there are two cases which, although they feel alike, are fundamentally different. When application code targets a library API, it does so usually to get a virtual copy of some abstractions for its own private use. To use this copy is not an act of communication; the application code still operates in a self-contained world, just one that is now augmented by whatever definitions the interface provides.

By contrast, we may be using an interface to communicate with some actual entity on the other side. Such an entity is not "part of the program"; it sits elsewhere. This is what C expressly facilitates in a first-class fashion. This distinction, about communication versus private abstraction, is not coincidentally reminiscent of the observations by William Cook [2009] about the pluralist, messaging-based style of data abstraction that is essential to objects and distinct from that offered by abstract data types.

(The contrast between hierarchical library-style abstractions and heterarchical communication seems to me remarkably similar to that between the lambda calculus—a system of computation via hierarchical, functional abstraction—and the pi calculus [Milner et al. 1992], a heterarchy of processes sharing an explicitly evolving communication structure. Although experts are quick to point out that one may easily encode one in the other, that does not erase the contrast in point-of-view that the two systems display.)

5.4 C versus tools: porosity and plurality

Allowing communicative code is not the only way in which C treats "the outside" differently. C implementations are distinctively cooperative with other tools, sitting alongside them at both build time (the assembler and linker) and at run time (the operating system and the system's debugging primitives). Each tool's boundaries with these neighbouring tools and systems are intentionally porous (consider `asm` statements, linker directives, etc.), and this allows allows further diverse and decoupled tools built upon them, exploiting commonality where it exists, and targeting mechanisms defined by the surrounding system (albeit non-portably) not by the language. In this way C acquires tool support without specifying its own tools "world" or ecosystem; the system, not the language, holds the power. It would be wrong to call a debugger like `gdb` a "C debugger"—by design, `gdb` mostly doesn't care what language you use, and its C front-end is a small and relatively inessential piece. Perhaps surprisingly, it does not work by "lowest common denominator" sharing of some fixed ABI; in fact it accepts a Turing-powerful descriptive language precisely in order to span arbitrary computational distances among disparate language implementations' internal choices. I wrote previously [Kell 2015] about this design at length, including the way certain core ideas were independently reinvented in the "mirrors" principles of Bracha and Ungar [2004]. That is not to defend `gdb` in detail, or the arcana of the technologies involved, which show all the negative effects of decades-long piecemeal evolution. The key idea is to avoid prescriptiveness in favour of a pluralist design. This can again be thought

of as hierarchy versus heterarchy: heterarchies concern existence "alongside", not "within" or "underneath".

Replicating these benefits in future languages is partly a cultural problem. For those who are used to thinking about designing or implementing *a language*, pluralism is a radical shift in mindset. This shift has much in common with the criticism by David Parnas [1978] of the premise of software development as concerning writing *a program*.

> Dijkstra's "Discipline of Programming" uses predicate transformers to specify *the* task to be performed by *the* program to be written. The use of the definite article implies that there is a unique problem to be solved and but one program to write. … The usual programming courses neither mention the need to anticipate changes nor do they offer techniques for designing programs in which changes are easy.

To accept pluralism is to break with the idea that infrastructure can or should be made of sealed boxes (whether languages or tools). The sealed-box endgame has each box's creators engaged in an endless squabble over which one gets to "win". The dominant implementation paradigm for high-level languages is the virtual machine—a monist conception to the core. Virtual machines, as implementation artifacts, realise a language as a box that is closed, save for tiny openings. A virtual machine seeks to *contain*, not coexist—except for minimal concessions in the form of the I/O and foreign function interfacing primitives.[10] The predictable squabble has been going on for decades.

To be pluralist, language implementations must change tack. They cannot encapsulate their every decision; they cannot stay opaque to the outside world. Instead they must proactively *describe* their own operation to the outside, in a form that is machine-readable and programmable against. Debugging metadata is the definitive example, and provides a strong and surprising ability, in the folklore of C and related languages' implementations, to decouple tools from language implementations.

One cost of this approach is the burden on the compiler and tool authors of establishing communication conventions, such as debugging information. Another hurdle is cultural. Porous boundaries among tools simply seem like anathema. Surely, following Parnas, we should not expose information, but hide it? Of course, diligent students of Parnas know that *hiding from whom* is everything; in this case, information is being exposed only to programs, as a parameter that may vary, not to humans as a detail to fix in code.

Until pluralist designs are adopted as a matter of course, every advance in language design creates a squabble, and unreasonably attempts to build a fresh world from scratch. This does not scale, and sticking with it dooms our programming infrastructure to the kind of immaturity and sluggish advancement that critics of C often lament.

5.5 Symbols and meanings

We have seen how C's abstraction for communicating is *memory*, a large collection of individually addressable communication channels. A channel has *symbols*, in the form of a representation of data. In C, that representation is manifest: the language actively permits programming against representations directly, including manipulating them (as bytes) as well as (separately) treating them with meaning (up to data types).

To illustrate the use of manifest representations, let's see yet another example. This one is about memory-mapped I/O—a very powerful primitive given to us by modern operating systems, for reasons both of efficiency (reducing copying, re-using buffering and swapping logic) and of programmability: what could be easier than manipulating data on disk exactly as if it were in memory? It is also, naturally, a communication feature. Not coincidentally, it is an abstraction which modern languages have continually failed to expose to their users, at least in its full power, but which is readily available from C.

The code in this case builds and manipulates an ELF file image. Rather than requiring separate serialisers and deserialisers to manage distinct on-disk and in-memory formats, the manifest nature of representations in C allows the language's own features—data types, defined with `struct`—to capture both the abstract and the concrete. They imply a particular bytewise layout both on disk and in memory.[11] The code exploits this to maintain ELF files in both places simultaneously, using a single pile of code, in a single style. Each file is created from a prototypical "zygote" in memory, by copying that memory into a memory-mapped region. Note that again, we also use linker features whose semantics are crucial—this time, to keep multiple structures contiguous in memory (the `section` directive; the unabridged code includes several other uses of this).

[10] By contrast, virtual machines *as a specification device* I have no problem with.

[11] Pedantic readers will note that the C language definition does not fix these details—although any C implementation will do so, via adherence to some platform-defined ABI standard. The code is non-portable with respect to differing application binary interfaces.

```
/* Declare a prototypical ELF header in memory. */
static Elf64_Ehdr ehdr
__attribute__((section(".elf_zygote")))
 = {
        .e_ident = { '\177', 'E', 'L', 'F', ELFCLASS64,
                ELFDATA2LSB, EV_CURRENT, ELFOSABI_GNU, 0 },
        .e_type = ET_DYN,
        .e_machine = EM_X86_64,
        .e_version = EV_CURRENT,
        .e_entry = 0,
        .e_phoff = (uintptr_t) &phdrs[0] - (uintptr_t) &ehdr,
        .e_shoff = (uintptr_t) &shdrs[0] - (uintptr_t) &ehdr,
        .e_flags = 0,
        .e_ehsize = sizeof (Elf64_Ehdr),
        .e_phentsize = sizeof (Elf64_Phdr),
        .e_phnum = PHDRS_N /* text, data, rodata, dynamic */,
        .e_shentsize = sizeof (Elf64_Shdr),
        .e_shnum = SHDRS_N /* ... */,
        .e_shstrndx = 1
};
/* snip more defs, also using .elf_zygote section */
void *_dlbind_elfproto_begin = &ehdr;
void *dlcreate(const char *libname)
{
        /* ... */
        void *addr = mmap(NULL, _dlbind_elfproto_memsz,
                PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
        if (addr == MAP_FAILED) goto out;
        /* Copy in the ELF proto */
        memcpy(addr, _dlbind_elfproto_begin,
                _dlbind_elfproto_stored_sz);
        /* ... */
}
```

A final interesting property of this code is that its behaviour is undefined according to the C language standard. The reason is that it calls memcpy() across a range of memory comprising multiple distinct C objects, copying them all into memory-mapped storage in a single operation. Without this the copy would have to be coded in multiple steps, reducing readability. Although C's memcpy() isn't obliged to support this case, real implementations generally do. I return to the thorny issue of undefined behaviour later (§7).

Contrast all this with a managed[12] language, where most likely, the best we can do is write serialisers and deserialisers between the byte-wise (disk) representation and the in-memory layout (objects of some unknown representation). There is no defined relationship between in-memory and on-disk layouts. Achieving the conciseness of the ELF format and the code which uses it therefore becomes impossible. Also, the primitive of memory-mapped I/O is necessarily curtailed; the managed runtime must be responsible for placing data in memory itself. It is easy to fix the first problem—by defining a language that is explicit about its data representation, but like other managed languages in other respects. Fixing the latter is harder, because in a managed language there are two kinds of memory: one is first-class memory, which is private to the language implementation and allows no communication. The other is the byte buffers we pass in and out of some magic narrow interface; these are memory that is communicated, but the operations they support are different from, and less expressive than, those of ordinary memory. Communication has become a second-class concern.

5.6 Going intergalactic

Meditating on this communicativity suddenly gave way to a realisation: C is designed for communicating with aliens!

J.C.R. Licklider conceived the ARPAnet as an "intergalactic network", and was paraphrased by Alan Kay as follows.

> "If we succeed in making an Intergalactic Network, then our main problem will be learning to communicate with Aliens."

Licklider was thinking about how software could function across displacements in space and time wide enough to find incompatible pre-existing "languages" on two sides of a newly formed communication link. This could be a network, or a programmatic interface—Licklider and his followers would probably not have seen a strong distinction between these.

In the case of C, I am clearly taking a few liberties. The "network" is smaller in scale—its scope is a single address space. And the "aliens" are not necessarily long-evolved distant neighbours—they are simply code or data that do not share the same host language implementation. The question of bootstrapping *meaning* is one for later—simply a *mechanism* for communication is the prerequisite. But an important aspect of the idea transfers: the C language (as specified in a book) and its implementations (in real programming systems) enshrine communicativity with these "not-known-to-me" entities. The ability to communicate is not an afterthought; it is essential and central to the design.

C and its implementations enshrine communicativity in three ways: by the compiler's place in a wider toolchain; by the surrounding meta-level (debugging) programmability of the target environment; and by the core abstraction of *memory* that is both at the heart of the language's design and shared with these surrounding elements. This is how C gives us access to the operating system and to hardware. The same property also gives us access to *other* systems in the same address space,

[12] This term is in popular use but is seldom defined. For now, a loose intuition suffices, which I assume the reader already possesses. However, I will return more closely to its definition in §6.

which is C finds so much use as a low-level glue language, despite its questionable suitability as such by any other criteria. Memory is a communication channel, shared with the world outside the language. Representations are the symbols of that channel. C lets us communicate freely using these, even with alien entities.

This is, I claim, the deepest reason why C remains unvanquished. Replacements or reimplementations invariably forgo or compromise communicativity. They break the links with the surrounding toolchain (particularly the assembler and linker), or provide a superficially similar but essentially different abstraction of memory. In so doing, they sacrifice its essential value as a systems programming language.

6 To manage or to mediate

To avoid the dangers of unfettered access to memory, high-level languages lock things down. User code may deal only with memory that the language implementation can account for. This is what I define to be the essence of "managed" environments: they manage memory, by restricting users to their own services for allocating, accessing and reclaiming it. While this admits runtime safety properties that help debugging and security, unfortunately, doing so also forgoes communicativity: we cannot write any of the code we just saw, which dealt with memory that was "alien". How do "managed" and "safe" really relate? I will argue that the former is a means to the latter end, but it is not the only possible means.

6.1 Managing what?

Our discourse uses "safe languages" and "managed languages" almost interchangeably. Conventional wisdom holds that if we want to be safe, we have to be managed. Safety is gained from control: of memory allocation, object layouts, of what pointers are created, and so on; by imposing strong invariants from above. The managed language lays down its vision of what memory is to hold, and enforces it, creating a tight seal.

"Managed", however, does not strictly imply "safe", since real managed language implementations invariably provide unsafe primitives—such as Java's infamous `sun.misc.Unsafe` in HotSpot, OCaml's `Obj.magic`, Haskell's (standard) `unsafePerformIO`, and so on. These may or may not be standard or even documented, but somehow they always turn out to merit inclusion.

Does the reverse hold—does "safe" imply "managed"? Or is there another means to make languages safe, perhaps without sacrificing a communicative memory abstraction? I believe there is, and will argue in particular that C could be implemented safely without losing its communicativity, albeit in a form that would look very different from a "managed" implementation.

6.2 What is safety anyway?

I have learned to enjoy provoking indignant incredulity by claiming that C can be implemented safely. It usually transpires that the audience have so strongly associated "safe" with "not like C" that certain knots need careful unpicking.

In fact, the very "unsafety" of C is based on an unfortunate conflation of the language itself with how it is implemented. Working from first principles, it is not hard to imagine a safe C.[13] As Krishnamurthi and Felleisen [1999] elaborated, safety is about catching errors immediately and cleanly rather than gradually and corruptingly. Ungar et al. [2005] echoed this by defining a safety property as "the behavior of any program, correct or not, can be easily understood in terms of the source-level language semantics"—that is, with a clean error report, not the arbitrary continuation of execution after the point of the error.

Let us consider the familiar type- and memory-safety property of (classic) Java[14]. A property rather like this is quite permissible by the C standard, because the operations which would corrupt the program state—out-of-bounds accesses, use-after-free, use of an incorrectly-typed pointer, and so on—necessarily have undefined behaviour. How corruption occurs is inherently implementation-dependent, so can never be described in a portable language specification. Being undefined does not preclude a clean trap; in fact, as I will argue later (§7), the design philosophy of Unix not-so-tacitly sanctions one if it can be provided. But such sanctioning does not belong in a portable language specification, because this assumption need not hold in all scenarios—such as in a tiny embedded device, or in a system simple enough that it can be proven free from undefined behaviours.

Consider unchecked array accesses. Nowhere does C define that array accesses are unchecked. It just happens that implementations don't check them. This is an *implementation norm*, not a fact of the language. Consider pointer arithmetic—almost synonymous with "unsafe", but actually just a nicety for array indexing. A pointer is not an iterator over raw memory; it has a type and a strong caveat, that it is anchored to the object in memory that it was created to point at. Overrunning into some other object has undefined behaviour. A clean trap of such overruns is both desirable and entirely permitted by the language.

[13] Conversely it is possible to imagine an uncommunicative implementation of C, which does not support any of the code snippets we saw earlier.

[14] This property took a while to become true—it was eventually fixed after it was pointed out that earlier versions of Java were "not type-safe" [Saraswat 1997].

At this point, my interlocutors are back on script. That sounds like fat pointers; how do you preserve binary compatibility? The answer is in the literature [Nagarakatte et al. 2009]. What about temporal safety? It's also in the literature, both on explicit checks over manual management [Nagarakatte et al. 2010] and on automatic management (the wealth of precise garbage collection literature, but also as applied to C [Necula et al. 2002; Rafkind et al. 2009]). What about all those obscure pointer tricks, like XORing them? These are not really pointer tricks so much as *address calculation* tricks, and happen in the integer domain not the pointer domain (recalling that pointers are anchored to the lvalue from which they were created). Since an address only becomes a pointer on the cast back, it can be checked at that point [Kell 2016]. What about memory scribbling via pointers to char (raw bytes)? Using scribbled-on data is mostly undefined, and for the cases when it isn't, one can insert checks *after* the scribbling and *before* use of the scribbled-on value. These latter two kinds of check are a bit tricky: they rely on some authority about *what's in memory*—something like run-time type information, allowing questions like "what's on the end of this pointer?" or "what are these bytes representing?". That, too, is in the literature, and the overheads are mostly low [Kell 2015]. This paragraph is a mixture of existing and hypothesised work (or, my sources tell me, work in progress) but I believe the essential point is not in doubt: a safe implementation of C is possible.

With all this checking, won't the result be slow? Certainly it will be slow*er*, although this is critical only if one believes (wrongly, as I have been claiming) that people use C for performance. At one time, Java "was slow", partly because of such dynamic checks, but the steady application of static reasoning techniques for eliminating them has essentially eliminated the problem. Java usually lags on memory grounds—consumption, predictability and cache interactions. A C with comparable dynamic checking would do better on these counts anyway, for example because C's more expressive options for placement and layout of structured data enable better memory locality.

Is such a safe implementation of C really suitable for systems programming, rather than merely application programming? If we understand system-building as communicativity, then certainly such a system retains communicativity—so long as alien objects can be *described* to it in a manner sufficient for dispatching the same dynamic checks. If I memory-map a file, say, I can safely access that memory only if the structure and meaning—the bounds and the types, roughly—are described much like those of other in-memory objects. Tools and systems for providing these descriptions are currently lacking—but are a logical extension of the run-time type information already developed in recent work. In the case of file formats, some cases like the ELF example we saw earlier (§5.5) show that the format has already been defined for us, thanks to the manifest layout of objects declared in C.

Once we have descriptions of memory, we also need descriptions of how different areas of memory relate. The natural generalisation of this idea is to *address-space contracts*, which I consider shortly (§6.5). A further detail is, of course, the extent to which they behave like "memory" at all—as evidenced, for example, by the device-register programming seen in Duff's device, and in general by the wider semantic envelope allowed of volatile objects.

If we mean systems programming in the sense of performance-critical low-level code, then I suspect the answer is "in some cases, but not all". The true answer remains to be seen; in-the-small performance is less critical than is often imagined, even to kernel hackers.

By this point I hope to have convinced the reader that the sort of implementation of C I have outlined offers a compelling path at least for application code that has *already* been written in C—much of which is code that "safe language" advocates have been telling us needs to be thrown away and rewritten.

6.3 You say "high-level"…

For systems programming within language runtimes, some prior literature has previously asked whether C and other "low-level" languages are appropriate, and concluded by advocating a variant of Java extended with unsafe primitives that can be selectively enabled [Frampton et al. 2009; Wimmer et al. 2013]. To me, this has always seemed to be answering a confused question. Of course, the resulting code still "looks like" Java, but what really defines "high-" or "low-"-level programming? Is a version of Java augmented with unchecked pointer values really still high-level? Is a dynamically checked C really still a low-level language? Convincing arguments are difficult in these terms. It seems preferable to level up—to make safe what was unsafe, rather than drilling careful holes, however small, in the hull of a previously safe ship—if all else is equal. Of course, at the sharp end of systems code, all else is never equal. In a world where compiler bugs are not uncommon—such as in kernels, thread schedulers or garbage collectors—language-derived guarantees are always suspect, and the question of safety becomes more about confidence in any particular implementation.

In fairness, there is one real benefit of the "it all looks like Java" approach, deriving from the re-use of tools and pedagogy accumulated around the Java language. C is certainly not a friendly language to learn, and not just

for reasons of unsafety. Its learning curve is steepened by a small standard library, often quirky syntax (such as "declaration reflects use" [Kernighan and Ritchie 1988, p. 122]), reliance on explicit indirection even in common cases (dynamic memory allocation, dynamic dispatch, passing by reference, and so on), laborious error-handling, and a culture of brevity that is sometimes extreme. Java-like systems programming environments can easily do better by these criteria, regardless of whether they are truly "higher-level".

6.4 A matter of performance

I have claimed already that even low-level code could make use of a safe, dynamically checked implementation of C. Although there is a cult of performance around the C language, it does not come from systems programmers, whose attitude to performance is generally holistic and led by design. If anything, extreme attention to performance comes from the opposite: attempts to generalise C towards specific application domains, particularly numerical applications. Linus Torvalds, no stranger to systems programming, is a persistent critic of language standards bodies and compiler implementers for pursuing small performance improvements at the expense of predictability and debuggability—as he expounded in characteristically direct style in a cross-mailing-list exchange in February 2016.

> …the original C designers were better at their job than a gaggle of standards people who were making bad crap up to make some Fortran-style programs go faster.

These issues come down to the interpretation of undefined behaviour in C—to which I will return briefly in §7.

6.5 Mediation, not management

So far I have outlined a C that has greater dynamic checking and the ability to accept descriptions—metadata of some kind—about the address space and the objects therein. Is that really enough, and does it constitute a fundamental break from the approaches of managed languages?

"Managed languages" is already a misnomer, because managed-ness is usually a property of an implementation, not a language. While one cannot rule out that a specification might preclude any other implementation approach, but this seems unlikely, given that language definitions attempt to give semantics to programs rather than specify how to achieve them. Languages may specify *safety*, which simply means that errors are trapped cleanly. But we don't *have* to specify safety properties in the language definition (as C elects not to); they can instead be a property of the implementation. Similarly, unsafety should not be confused with error-proneness. Safety *can* be achieved by having the language implementation exclude all but a set of necessary and fully-defined operations—but it doesn't have to be. It might instead support a wider set of operations, including error-prone ones (like manual storage management), but still ensuring that errors are trapped when they occur.

Fundamentally, a safety guarantee requires quantifying over all possible scenarios. It means somehow "closing the box"—circumscribing the domain of quantification. It is only if we insist on a safe *language*, as opposed to a safe implementation of a not-specified-as-safe language, that this box-closing trick has to happen inside the language definition. Since language definitions are usually intended to be portable, this means our box-closing trick must be *generic with respect to the host environment*.

The rest is history. As we know, the way this requirement is satisfied, in managed languages, is to drastically restrict communication with that host environment, usually down to some generic "file" or "channel" operations that deal in byte buffers. The role of these is to provide a narrow, opaque interface that can be laid down in the language definition.

But we can reject the idea of safety as a property specified in languages. It can instead be a property that the surrounding system establishes for us. If we do this, the language specification itself, as a portable document divorced from the system, necessarily becomes somewhat underspecified. Conversely, we need to refine the role of the wider system, into one that can *mediate* between the diverse objects that might appear in an address space, and the code that would access them.

An operating system necessarily provides facilities by which separate units of software communicate. By and large, these map to memory in some fashion. Safety may therefore be bootstrapped partly within the design of the operating system services that deal in memory.

In a typical modern operating system these are the virtual memory subsystem (on the kernel side; activated by run-time system calls) and the linker (on the user side; largely activated at load time, in the dynamic linker). However, equivalent services exist even where there is no surrounding operating system, such as in embedded targets or hypervisors: there is still an ahead-of-time link step that makes these decisions, together with some stuff that configures the initial environment, whether a BIOS, bootloader, ROM programmer, dipswitch array, or rat's nest of jumper cables. In these cases, the surrounding system is defined by the hardware platform rather than an operating system, but the principle is the same. So long as there is C code to be written, there is a notion of address space and a system for managing it.

Communicative code makes assumptions about what lies within the address space, and defines its own behaviour in a way that is conditional on those assumptions. Given an adequate specification for code's surroundings, we can bootstrap safety properties *on top* of the language definition, *rather than within it*.

Consider the code snippets we surveyed earlier: each's correctness depended on some property of the address space layout in relation to its own memory. Not coincidentally, the idea of "assume-guarantee" reasoning—reasoning compositionally about pieces of a whole using mutually conditionalised properties—is well established in software verification. The same ideas in on-line reasoning, specifically dynamic checking, are familiar from the extensive literature on *contracts*, originating as simple assertions but now generalised considerably [Findler and Felleisen 2002].

Contracts can be said to *mediate* interactions. Contracts, and similar styles of conditional reasoning, seem to offer an alternative and general basis for safety properties (and indeed correctness properties) of unmanaged code. This is not a technical paper, so, conveniently perhaps, is not the place to develop any details in the idea of address space contracts. But much as communication demands a shift from hierarchy to heterarchy, these concepts of mutual conditionality turn units of code and data into reciprocally "mediating" entities—a natural and stark contrast to the unilaterality of "managed" environments, in which the desired properties are imposed by fiat from above. Rather than levelling down by sacrificing communication, there is reason to believe that we can level up, by finding inherently compositional ways of establishing the properties we seek. *Language-level* safety properties become simply a degenerate case: properties that hold under a generic, controlled, alien-free environment.

6.6 Safety and the meta-level

Implementations of C are typically, in certain ways, metacircular. That does not mean that they are themselves written in C, although they can be. As noted by Chiba et al. [1996], in a true metacircular system each turn around the "circle" is actually advancing *the same* system in some orthogonal dimension, adding some feature, service or behaviour that the base system lacked. (It follows that the "same language" property of a metacircular interpreter is a relatively superficial and uninteresting one, and neither necessary nor sufficient for a metacircular system. A self-hosting compiler, say, is not meaningfully metacircular, because the compiling compiler is not reused or extended by the compiled compiler—they simply happen to be expressed in the same language.)

Most obviously in C, we note that a `malloc()` implementation is usually written in C—or rather, in a subset of C that lacks `malloc()` since `malloc()` is mandated by the C standard. Doing this requires invoking C's "porosity", noted earlier (§5.4), with respect to the surrounding system, typically by using the assembler to issue system calls. In so doing, both "base" (`malloc()`-less) and "extension" (`malloc()`-enabled) Cs exist within the same system, modelling a kind of metacircular extension. (Of course, the assembly code is given no semantics by the C language specification, but gains its meaning from the specifications of the machine and the assembler.) A more extreme example is a Unix-style dynamic loader, whose initial routines are typically written in a subset of C restricted (also using knowledge of the compiler) so as to use only stack- and program counter-relative addressing modes, because the loader's own program text has not yet been relocated (it must do that itself!).

Our instrumentation code snippet (§5.2) is another example of metacircular extension—here used to add services via a trap-and-emulate layer over the instruction stream. By instrumenting the base program, yet excluding itself, it is working with two implicit meta-levels: its own, which (to avoid infinite regress) must do without the added service, and the user code's, to which that service is added. C does not define a "reflection system" per se, but the language's core abstraction—a communicative memory—naturally allows code to reflect on its own implementation, which is found in memory like other objects (here as stored instruction and data objects). The result is in some sense *more powerful* than many a reflective language's built-in facilities. This ability to employ one's own abstractions in an act of self-extension is the essence of metacircularity as an engineering technique.

Doing so is, of course, dangerous—and my reason for mentioning all this relates to safety. A naïve view might be that establishing memory safety for code such as a `malloc()` implementation is impossible, given that the code is responsible for providing the very abstraction in whose terms we would like to state our safety properties—about allocated chunks and their boundaries. But in fact, the `malloc()` code is simply C code working lower down the helix, on a different view of the same memory. There, too, safety properties may be stated and established: the `malloc()` should only write into its own arena, say, not some wholly unrelated area of memory. The helical view, of distinct meta-levels at which distinct properties are sought, promises a much deeper set of "safety" notions appropriate for code residing at different places in the helix. My own infrastructure models an "allocator hierarchy" [Kell 2015] which appears already to capture much of this relation in the specific case of allocators.

7 An alternative history of undefinedness

Brian Kernighan, although not a designer of the C language, is perhaps its most eloquent exponent, co-authoring the classic text on the language [Kernighan and Ritchie 1988]. He is also known for a debugging aphorism [Kernighan and Plauger 1978].

> "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"

Achieving debuggability and reliability through simplicity and comprehensibility was central to the Unix philosophy. Somehow, modern implementations of C have diverged utterly from this. In the words of Linus Torvalds, they are adept at "turning [a] small mistake into a real and exploitable security hole". His complaint was directed at compiler authors for their eagerness to exploit *undefined behaviour* in the C language. This makes simple programming errors difficult to detect, and silently escalates them from what might be clean crashes into dangerous continuations of execution, often with security consequences. C compilers try to be as clever as possible, at the expense of simplicity and debuggability.

Undefined behaviour mostly appears in the C standard as statements of the form "if *cond*, the behaviour is undefined". Here, *cond* is an error condition: only incorrect programs allow it to occur. A compiler exploits this when reasoning about code by inserting (at least conceptually) the proposition "assume not *cond*" as an input to its reasoning engine.

This assumption typically enables code transformation. For example, Chris Lattner [2011] explains optimisations based on undefined behaviour using the following simple example.

```
void contains_null_check(int *P) {
  int dead = *P;
  if (P == 0)
    return;
  *P = 4;
}
```

Since dereferencing a null pointer, even from otherwise dead code, is undefined, the compiler can assume that the pointer is *not null*, admitting arbitrary consequences in the cases where this assumption is false. "Arbitrary consequences" is the chosen interpretation of "undefined behaviour". In this case it allows all but the last statement to be deleted, which in a less trivial example would cause execution of code that the preceding check intended to skip.

The arbitrary is the enemy of the debuggable. Undefined behaviour appears to be a joke in poor taste: it works so thoroughly against debuggability as to make Kernighan and colleagues seem thorough hypocrites.

Fortunately there is another explanation. Behaviour is left undefined not to allow optimisation, but as an inevitable consequence of two things. Firstly, the program containing it is an error—the language implementation is not required to support it. Secondly, *no behaviour for detecting or handling the error can be prescribed in all cases*.

What *should* happen is a matter for implementations; it is impossible, in the language specification, to anticipate the details of the implementation scenario. So, *no particular* behaviour can be specified. In fact, if only *one* valid implementation scenario—perhaps the tiny embedded device, or a system for running only verified-correct code—can admit corrupting failure or other arbitrary consequences, then the language specification must allow this. And so the behaviour must be left undefined.[15]

This is not say that these behaviours could instead have been treated as implementation-defined. Invoking an implementation-defined behaviour is emphatically *not* an error, since the language implementation is obliged to handle this in some consistent fashion. It would be perverse if implementations could "define as undefined" these behaviours, subverting the intention of legitimate implementation-defined behaviours (such as whether right-shifting a signed number propagates its sign bit rightwards) and making the construct meaningless. Therefore, any condition that is properly an unrecoverable error—like dereferencing a null pointer—simply cannot become implementation-defined behaviour. There are a few grey areas, particularly with arithmetic operations where undefinedness could arguably be downgraded. But the need for both mechanisms is not in doubt.

Compiler writers take the opportunity to "assume not *cond*". A non-identical but very similar effect on optimisation can be had by instead inserting a test: "if *cond*: abort". Both ways give us a postcondition in which *cond* is not true, and let us optimise subsequent code accordingly—albeit at the cost of evaluating *cond* at run time, which may not be easy or efficient to implement. The vast dynamic-checking literature is concerned with exactly this: inserting just enough checks of this kind (or, sometimes, a non-aborting kind) while widening the space of efficiently evaluable conds (so as to include things like run-time bounds or types).

[15] This suggests the possibility of a stratified language specification, with stronger properties for implementations targeting conventional execution environments. This would be feasible, and indeed research on precise specification of "de facto C" suggests a move in this direction [Memarian et al. 2016]. However, standard bodies remain to be persuaded into accepting this step-change in complexity.

Torvalds himself mentions[16] the possibility of extra code yielding the same fast-path optimisations without the surprises of undefined behaviour.

> "Have you ever seen code that cared about signed integer overflow? Yeah, getting it right can make the compiler generate an extra ALU instruction once in a blue moon, but trust me—you'll never notice. You *will* notice when you suddenly have a crash or a security issue due to bad code generation, though.

When contradicted with the example of loop vectorisation, he continues.

> "It would generally force the compiler to add a few extra checks when you do vectorize (or, more generally, do any kind of loop unrolling), and yes, it would make things slightly more painful. You might, for example, need to add code to handle the wraparound and have a more complex non-unrolled head/tail version for that case."

Finally he notes a difference in attitude between compiler writers and system builders.

> "Performance doesn't come from occasional small and odd micro-optimizations. I care about performance a lot, and I actually look at generated code and do profiling etc. None of those three options have *ever* shown up as issues. But the incorrect code they generate? It has."

Today's compiler authors are reluctant to generate extra code: this complicates their task, and for what reason? From a language standpoint alone, there is none; the slightly faster and shorter code is better according to all available metrics (speed and size), and remains within the letter of the language. But this primacy of the language specification is a cultural issue, not a necessary consequence of that specification's existence.

Compiler authors are quick to claim that the performance penalty of *not* exploiting undefined behaviour would be significant. This claim may or may not be true; to my knowledge, evidence either way is lacking. Certainly, if we turn off all the optimisations which currently exploit undefined behaviour, the generate code is slower. The question is rather about whether, if we declined to exploit undefined behaviour for optimisation, we could implement a non-identical but competitive selection of optimisations—just *differently*, such as Torvalds describes. This might include extra tests, giving the same postcondition for later optimisation but branching to clean aborts and/or "carry on" slow paths.

The claim that doing so would be inexcusably inefficient is, at best, in need of justification. It is plausible that code size increases would cause problems for embedded code, but even the tiniest embedded devices now seem beefy (even to those with short memories). On commodity platforms, the argument seems untenable: spare instruction-level parallelism means extra ALU instructions cost very little; and the code size penalty is also unlikely to be significant. Of course, this remains speculation until such reimplemented optimisations exist.

It would be unfair not to acknowledge the value of various recent compiler improvements for detecting undefined behaviour, such as LLVM's UBSan tool. My critique of these approaches would be that they detect problems but do not, by themselves, fix them. They also rely too much on disciplined opt-in; there is still no safe default. This matters for the common case where old "believed correct" code is recompiled with a newer compiler, whose new optimisations open up bugs or security vulnerabilities. One "obvious" technical escape route would be to make the extra optimisations, not the checking, the opt-in.

In summary, undefined behaviour as a specification device is essential for portability, but its use as an enabler of optimisations is distinct—certainly not as inevitable, obvious or traditional as some claim. It is odd that it could be so ingrained in the culture of compiler writers, despite being so far from the original spirit of C. I suggest again that languages have gained undue primacy—what is written in the language specification is all that matters, even though the end to which the language is used, namely system-building, has other considerations.

8 Conclusions

I have argued that C's enduring popularity is wrongly ascribed to performance concerns; in reality one large component of it (the "application" component) owes to decades-old gaps in migration and integration support among proposed alternatives; another large component of it (the "systems" component) owes to a fundamental and distinctive property of the language which I have called its *communicativity*, and for which neither migration nor integration can be sufficient. I have also argued that the problems symptomatic of C code today are wrongly ascribed to the C language; in reality they relate to its implementations, and where for each problem

[16] … in a message dated 2016/2/28, cross-posted to various lists, available at https://gcc.gnu.org/ml/gcc/2016-02/msg00381.html as retrieved on 2017/7/13.

the research literature presents compelling alternative implementation approaches. From this, many of the orthodox attitudes around C are ill-founded. There is no particular need to rewrite existing C code, provided the same benefit can be obtained more cheaply by alternative implementations of C. Nor is there a need to abandon C as a legitimate choice of language for new code, since C's distinctive features offer unique value in some cases. The equivocation of "managed" with "safe" implementations, and indeed the confusion of languages with their implementations, have obscured these points.

Rather than abandoning C and simply embracing new languages implemented along established, contemporary lines, I believe a more feasible path to our desired ends lies in both better and materially *different implementations* of both C and non-C languages alike. These implementations must subscribe to different principles, emphasising heterarchy, plurality and co-existence, placing higher premium on the concerns of (in application code) migration and interoperation, and (in the case of systems code) communicativity. My concrete suggestions—in particular, to implement a "safe C", and to focus attention on communicativity issues in this and any proposed "better C"—remain unproven, and perhaps serve better as the beginning of a thought process than as a certain destination. C is far from sacred, and I look forward to its replacements—but they must not forget the importance of communicating with aliens.

Acknowledgments

I thank Philipp Haller and Paolo Giarrusso for their part in a formative conversation, Stephen Dolan for his part in several others, Peter Sewell and Jon Crowcroft for comments on earlier versions of the text, and (last but not least) the anonymous reviewers for their patient and highly constructive feedback. This work was supported by EPSRC grant EP/K008528/1, "Rigorous Engineering for Mainstream Systems".

References

AT&T. 1990. *UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools*. AT&T, Upper Saddle River, NJ, USA.

Per Bothner. 2003. Compiling Java with GCJ. *Linux Journal* (2003).

Gilad Bracha and David Ungar. 2004. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '04)*. ACM, New York, NY, USA, 331–344. https://doi.org/10.1145/1028976.1029004

CERT. 2014. OpenSSL TLS heartbeat extension read overflow discloses sensitive information. Vulnerability Note VU#720951. (2014). https://www.kb.cert.org/vuls/id/720951 as retrieved on 2017/8/28.

Shigeru Chiba, Gregor Kiczales, and John Lamping. 1996. Avoiding confusion in metacircularity: The meta-helix. In *Object Technologies for Advanced Software: Second JSSST International Symposium*, Kokichi Futatsugi and Satoshi Matsuoka (Eds.). Springer, Berlin, Heidelberg, 157–172. https://doi.org/10.1007/3-540-60954-7_49

William R. Cook. 2009. On Understanding Data Abstraction, Revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 557–572. https://doi.org/10.1145/1640089.1640133

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/581478.581484

Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. 2009. Demystifying Magic: High-level Low-level Programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '09)*. ACM, New York, NY, USA, 81–90. https://doi.org/10.1145/1508293.1508305

Richard P. Gabriel. 1994. Lisp: Good News, Bad News, How to Win Big. *AI Expert* 6 (1994), 31–39.

Richard P. Gabriel. 2012. The Structure of a Programming Language Revolution. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '12)*. ACM, New York, NY, USA, 195–214. https://doi.org/10.1145/2384592.2384611

Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. 2013. An Efficient Native Function Interface for Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13)*. ACM, New York, NY, USA, 35–44. https://doi.org/10.1145/2500828.2500832

Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. 2011. Passing a Language Through the Eye of a Needle. *Commun. ACM* 54, 7 (July 2011), 38–43. https://doi.org/10.1145/1965724.1965739

Trevor Jim. 2015. C doesn't cause buffer overflows, programmers cause buffer overflows. Blog article. (2015). http://trevorjim.com/c-doesnt-cause-buffer-overflows--programmers-cause-buffer-overflows/ as retrieved on 2017/8/28.

Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, Berkeley, CA, USA, 275–288. http://dl.acm.org/citation.cfm?id=647057.713871

Alan Kaplan, John Ridgway, and Jack C. Wileden. 1998. Why IDLs Are Not Ideal. In *Proceedings of the 9th International Workshop on Software Specification and Design (IWSSD '98)*. IEEE Computer Society, Washington, DC, USA, 2–7. http://dl.acm.org/citation.cfm?id=857205.858288

Stephen Kell. 2015. Towards a Dynamic Object Model Within Unix Processes. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA, 224–239. https://doi.org/10.1145/2814228.2814238

Stephen Kell. 2016. Dynamically Diagnosing Type Errors in Unsafe Code. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 800–819. https://doi.org/10.1145/2983990.2983998

Stephen Kell, Dominic P. Mulligan, and Peter Sewell. 2016. The Missing Link: Explaining ELF Static Linking, Semantically. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 607–623. https://doi.org/10.1145/2983990.2983996

Brian W. Kernighan and Phillip James Plauger. 1978. *The elements of programming style*. McGraw-Hill.

Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language* (second ed.). Prentice Hall.

Shriram Krishnamurthi and Matthias Felleisen. 1999. *Safety in programming languages*. Technical Report TR 99-352. Rice University.

Chris Lattner. 2011. What Every C Programmer Should Know About Undefined Behavior. Blog article (in three parts). (2011). http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_21.html as retrieved on 2017/7/13 (third part, containing links to others).

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the De Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/2908080.2908081

Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes, I. *Information and Computation* 100 (1992), 1–40.

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI '09)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM '10)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/1806651.1806657

George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, New York, NY, USA, 128–139. https://doi.org/10.1145/503272.503286

David L. Parnas. 1978. Designing Software for Ease of Extension and Contraction. In *Proceedings of the 3rd International Conference on Software Engineering (ICSE '78)*. IEEE Press, Piscataway, NJ, USA, 264–277. http://dl.acm.org/citation.cfm?id=800099.803218

Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. 2009. Precise Garbage Collection for C. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09)*. ACM, New York, NY, USA, 39–48. https://doi.org/10.1145/1542431.1542438

The Santa Cruz Operation, Inc. 1997. System V ABI specification, edition 4.1. (1997).

Vijay Saraswat. 1997. Java is not type-safe. Web note. (1997). http://www.cis.upenn.edu/~bcpierce/courses/629/papers/Saraswat-javabug.html as retrieved on 2017/8/28.

Claude E. Shannon and Warren Weaver. 1949. *A Mathematical Theory of Communication*. University of Illinois Press.

Simon Tatham. 2000. Coroutines in C. Web page. (2000). http://www.chiark.greenend.org.uk/%7esgtatham/coroutines.html as retrieved on 2017/8/28.

David Ungar, Adam Spitz, and Alex Ausch. 2005. Constructing a metacircular Virtual machine in an exploratory programming environment. In *Companion to the 20th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/1094855.1094865

Philip Wadler. 1998. Why No One Uses Functional Languages. *SIGPLAN Not.* 33, 8 (Aug. 1998), 23–27. https://doi.org/10.1145/286385.286387

Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (Jan. 2013), 24 pages. https://doi.org/10.1145/2400682.2400689