Some were meant for C

The endurance of an unmanageable language

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge
<table>
<thead>
<tr>
<th>No.</th>
<th>Drink</th>
<th>ABV</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Partido</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Black Sheep</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Boekje Wonderland</td>
<td>6.5%</td>
<td>5.10</td>
</tr>
<tr>
<td>4</td>
<td>Jeune Pils</td>
<td>4.3%</td>
<td>3.80</td>
</tr>
<tr>
<td>5</td>
<td>Westmalle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Weisse SG</td>
<td>6.0%</td>
<td>5.50</td>
</tr>
<tr>
<td>7</td>
<td>Good Morning Sunshine</td>
<td>3.8%</td>
<td>5.20</td>
</tr>
<tr>
<td>8</td>
<td>La Trappe Blond</td>
<td>6.2%</td>
<td>4.20</td>
</tr>
<tr>
<td>9</td>
<td>Pé Biercollective, Prominent 2016</td>
<td>6.3%</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Gallopolo</td>
<td>15.8%</td>
<td>10.50</td>
</tr>
<tr>
<td>11</td>
<td>St. Peter's Organic Bitter</td>
<td>4.1%</td>
<td>4.</td>
</tr>
<tr>
<td>12</td>
<td>Wallahla Demon #1 B.I.A.</td>
<td>9.2%</td>
<td>8.20</td>
</tr>
<tr>
<td>13</td>
<td>Hommelies Porken Pumpkin Ale</td>
<td>7%</td>
<td>5.20</td>
</tr>
<tr>
<td>14</td>
<td>Anchor Wild Princess</td>
<td>4.2%</td>
<td>4.40</td>
</tr>
<tr>
<td>15</td>
<td>Wild Beer Co. Inde Wildeman Brett Pale Ale</td>
<td>4%</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Art Bier Van Demaand Oatmeal Stout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Cuitje Schwarzwalderkirschen</td>
<td>8%</td>
<td>4.50</td>
</tr>
</tbody>
</table>
It’s 2017. Surely people shouldn’t be programming in C?
It’s 2017. Surely people shouldn’t be programming in C?

- it’s outdated
It’s 2017. Surely people shouldn’t be programming in C?

- it’s outdated
- it’s inexpressive
It’s 2017. Surely people shouldn’t be programming in C?

- it’s outdated
- it’s inexpressive
- it’s unsafe!
It’s 2017. Surely people shouldn’t be programming in C?

- it’s outdated
- it’s inexpressive
- it’s unsafe!

So why do they? It must be…
It’s 2017. Surely people shouldn’t be programming in C?

- it’s outdated
- it’s inexpressive
- it’s unsafe!

So why do they? It must be…

- “performance!”
It’s 2017. Surely people shouldn’t be programming in C?

- it’s outdated
- it’s inexpressive
- it’s unsafe!

So why do they? It must be . . .

- “performance!”
- “old habits!”
It’s 2017. Surely people shouldn’t be programming in C?

- it’s outdated
- it’s inexpressive
- it’s unsafe!

So why do they? It must be...

- “performance!”
- “old habits!”
- “malign intent!”
I often program in C. For what reasons?
I often program in C. For what reasons?

- not performance
I often program in C. For what reasons?

- not performance
- not because I don’t know other languages
I often program in C. For what reasons?

- not performance
- not because I don’t know other languages
- not because I’m evil
I often program in C. For what reasons?

- not performance
- not because I don’t know other languages
- not because I’m evil
- sometimes *systems programming*
I often program in C. For what reasons?

- not performance
- not because I don’t know other languages
- not because I’m evil
- sometimes *systems programming*
- sometimes *integration purposes*
I often program in C. For what reasons?

- not performance
- not because I don’t know other languages
- not because I’m evil
- sometimes *systems programming*
- sometimes *integration purposes*
- often… enjoyment? (should I feel bad?)
C is bad and you should feel bad if you don’t say it is bad

May 23, 2016

I’ve spent a lot of time on this blog pointing out how C and C++ are to blame for most of the severe computer security failures we see on a daily basis. The evidence so overwhelming (and well known!) that in my experience even the most rabid C partisans do not challenge it.
“People use C because it’s faster, not for expressiveness.”

“C code is unsafe code.”

“The worst thing about C is all that undefined behaviour.”
“People use C because it’s faster, not for expressiveness.”

- C is peculiarly expressive w.r.t. communication

“C code is unsafe code.”

“The worst thing about C is all that undefined behaviour.”
“People use C because it’s faster, not for expressiveness.”

- C is peculiarly expressive w.r.t. communication

“C code is unsafe code.”

- unsafety of C is an implementation choice

“The worst thing about C is all that undefined behaviour.”
“People use C because it’s faster, not for expressiveness.”

- C is peculiarly expressive w.r.t. communication

“C code is unsafe code.”

- unsafety of C is an implementation choice

“The worst thing about C is all that undefined behaviour.”
“People use C because it’s faster, not for expressiveness.”
  - C is peculiarly expressive w.r.t. communication

“C code is unsafe code.”
  - unsafety of C is an implementation choice

“The worst thing about C is all that undefined behaviour.”
  - UB has sound motivations! problem: exploitation
Those claims again:

1. C is peculiarly expressive w.r.t. communication
2. unsafety of C is an implementation choice
3. UB has sound motivations! problem: exploitation
Those claims again:

1. C is peculiarly expressive w.r.t. communication
2. unsafety of C is an implementation choice
3. UB has sound motivations! problem: exploitation
Some things I’m not claiming:

- the C language is wonderful
- language wars / fanboyism are worthwhile
- anything about C++

When we/I say “C”, we might mean

- the C language as specified
- the C language as it is typically implemented
- the experience of using C implementations, in context
C was designed for ‘systems code’; means *communication*!

- with particular CPU
- with particular hardware
- with *non-C* code (assembly, ...)
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);
  }
}
Programs abstract over memory

- program memory
- alien (extralinguistic) memory

The same expressions (lvalues) abstract over both!

Alien memory is “first-class”!
Some ‘systems’ features in C as commonly implemented…

- first-class alien memory
- asm, attributes, … syntactic porosity
- explicit representations
- non-memory-like memory
- explicit address space
static Elf64_Ehdr ehdr SECTION(".elf_zygote");

// ...

ehdr = (Elf64_Ehdr) {
    .e_ident = { '\177', 'E', 'L', 'F', ELFCLASS64,
                ELFDATA2LSB, EV_CURRENT, ELFOSABI_GNU, 0 },
    .e_type = ET_DYN,
    .e_machine = EM_X86_64,
    /* ... */
};

// ...

void *addr = mmap(NULL, sz, PROT_READ|PROT_WRITE,
                   MAP_SHARED, fd, 0);
if (addr == MAP_FAILED) goto out;
/* Copy in the ELF proto */
memcpy(addr, ehdr, sz);
// ...
symbols in

channel

symbols out
Meaning is communicated in discrete, manifest units

- in communication theory: “symbols” from an alphabet
- in C: representations manifest as bytes

```c
float f = PI;
fwrite(&f, sizeof f, 1, stdout);
int i; memcpy(&i, &f, sizeof f);
```
Memory can be channel-like too...

```c
unsigned count = ...;
do {
    *to = *from++;
} while(--count > 0);
```
Channels are numbered, forming an *address space*
uintptr_t upper = next_obj_loaded_above(stackptr)->l_addr;
for (const char **p_str = &environ[0]; *p_str; ++p_str)
{
    if (((uintptr_t) *p_str > lower
        && (uintptr_t) *p_str < upper)
    {
        /* It's pointing into the auxv's environ block.
         * Search for the adjoining auxv structures. */
        ElfW(auxv_t) *searchp
            = ((uintptr_t) *p_str) & ~(sizeof(void*) - 1);
        while (!((/* complex "found it" cond */ )))
        {
            searchp = /* next address to inspect */;
        }
        /* assert we found it */
        assert (searchp->a_type == AT_NULL
            && !searchp->a_un.a_val);
    }
}
Summary: ‘communicative’ language [impls] have

- first-class alien memory
- explicit representations
- non-memory-like memory
- computation over address space
Summary: ‘communicative’ language [impls] have

- first-class alien memory
- explicit representations
- non-memory-like memory
- computation over address space

Contrast: managed runtimes

- memory is private $\rightarrow$ not for communication
- … use separate magic primitives (I/O, FFI) instead
- memory is memory (e.g. no ‘native fields’)
- representations are hidden
- no addresses, just opaque identity
Communicativity not unique to C! Subsets found in

- (C++ of course)
- some impls of Pascal-family languages
- unsafe Modula-3
- unsafe Rust...
Code is ‘application code’ if it doesn’t need this…

Little reason to use C for new application code[bases]

- … but also huge cost to rewriting older C
- high-level VMs don’t support gradual rewrite…
- + embrace use of C as a glue language (horribly)

The above is only mostly true…

- some affordances, like mmap(), are widely useful
1. C is peculiarly expressive w.r.t. *communication*
2. unsafety of C is an *implementation choice*
3. UB has sound motivations! problem: *exploitation*
1. C is peculiarly expressive w.r.t. communication

2. unsafety of C is an implementation choice

3. UB has sound motivations! problem: exploitation
We usually see *safety* as a property of languages.
We usually see safety as a property of languages.

“A language is safe (1) if it precisely specifies the set of data for which its operations are defined and (2) if it signals an error when an operation is applied to inappropriate data. . .

“The very fact that safe languages do signal errors is their key advantage.”

–Krishnamurthi & Felleisen
“Safety in programming languages”
Rice COMP TR99-352, 1999
Can instead view safety as a property of *evaluators*...
Can instead view safety as a property of *evaluators* . . .

“A [program] that works properly has the same behavior on both [unsafe and safe] evaluators. Unfortunately, for a program that misinterprets data we *cannot predict* how it behaves under the unsafe evaluator.”
Can instead view safety as a property of evaluators…

“A [program] that works properly has the same behavior on both [unsafe and safe] evaluators. Unfortunately, for a program that misinterprets data we cannot predict how it behaves under the unsafe evaluator.”

Safety is always about predictability

- whether static or dynamic
Can instead view safety as a property of *evaluators*…

“A [program] that works properly has the same behavior on both [unsafe and safe] evaluators. Unfortunately, for a program that misinterprets data we *cannot predict* how it behaves under the unsafe evaluator.”

Safety is always about predictability

- whether static or dynamic

Can we have a “safe evaluator” for C?
The infamously unsafe pointer arithmetic:

```c
int *p = /* ... */;
int *q = p + off;  // if this arithmetic is bad,
*q = /* ... */;  // anything could happen!
```
The infamously unsafe pointer arithmetic:

```c
int *p = /* ... */;
int *q = p + off;  // if this arithmetic is bad,
*q = /* ... */;   // anything could happen!
```

Claim: this is ‘unsafe’, because...

If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary * operator that is evaluated.
Imagine a “safe” implementation that translates to:

```c
int *p = /* ... */;
int *q = SAME_OBJECT(p, p+off) ? p + off : TRAP(p + off);
*q = /* ... */; // traps if our arithmetic was bad
```
Imagine a “safe” implementation that translates to:

```c
int *p = /* ... */;
int *q = SAME_OBJECT(p, p+off) ? p + off : TRAP(p + off);
*q = /* ... */; // traps if our arithmetic was bad
```

Is this some kind of new dialect of C? Perhaps a subset of C?
Imagine a “safe” implementation that translates to:

```c
int *p = /* ... */;
int *q = SAME_OBJECT(p, p+off) ? p + off : TRAP(p + off);
*q = /* ... */; // traps if our arithmetic was bad
```

Is this some kind of new dialect of C? Perhaps a subset of C?

- same feature set, same code, same idioms
Imagine a “safe” implementation that translates to:

```c
int *p = /* ... */;
int *q = SAME_OBJECT(p, p+off) ? p + off : TRAP(p + off);
*q = /* ... */; // traps if our arithmetic was bad
```

Is this some kind of new dialect of C? Perhaps a subset of C?

- same feature set, same code, same idioms
- same set of valid programs
Imagine a “safe” implementation that translates to:

```c
int *p = /* ... */;
int *q = SAME_OBJECT(p, p+off) ? p + off : TRAP(p + off);
*q = /* ... */; // traps if our arithmetic was bad
```

Is this some kind of new dialect of C? Perhaps a subset of C?

- same feature set, same code, same idioms
- same set of valid programs
- same set of behaviours in error-free executions
Imagine a “safe” implementation that translates to:

```c
int *p = /* ... */;
int *q = SAME_OBJECT(p, p+off) ? p + off : TRAP(p + off);
*q = /* ... */; // traps if our arithmetic was bad
```

Is this some kind of new dialect of C? Perhaps a subset of C?

- same feature set, same code, same idioms
- same set of valid programs
- same set of behaviours in error-free executions
- smaller set of behaviours on error
Imagine a “safe” implementation that translates to:

```c
int *p = /* ... */;
int *q = SAME_OBJECT(p, p+off) ? p + off : TRAP(p + off);
*q = /* ... */; // traps if our arithmetic was bad
```

Is this some kind of new dialect of C? Perhaps a subset of C?

- same feature set, same code, same idioms
- same set of valid programs
- same set of behaviours in error-free executions
- smaller set of behaviours on error

Not a subset or dialect. It’s a refinement!
The C language is customarily refined

- `man cc`
- ABI specifications
- norms of hardware, OS, vendor, …

Semantically, our ‘safe’ refinement of C

- is a tighter semantic envelope
- is relatively peripheral / separable
- *supplements* the language spec

Complete “freedom to refine” is called *undefined behaviour*

- good or bad? well…
1. C is peculiarly expressive w.r.t. communication
2. unsafety of C is an implementation choice
3. UB has sound motivations! problem: exploitation
Intro to undefined behaviour, due to John Regehr

```c
#include <limits.h>
#include <stdio.h>

int main (void)
{
    printf ("%d\n", (INT_MAX+1) < 0);
    return 0;
}
```
Intro to undefined behaviour, due to John Regehr

```c
#include <limits.h>
#include <stdio.h>

int main (void)
{
    printf ("%d\n", (INT_MAX+1) < 0);
    return 0;
}

$ cc test.c -o test
$ ./test
Formatting root partition, chomp chomp
Why might a language spec leave behaviour undefined?

- “to enable optimization”—a popular story
- to enable a *portably efficient language specification*
Modern C is specified to be broadly *portable*. 
Modern C is specified to be broadly portable.

Subtlety: does not mean that programs need be portable!
Modern C is specified to be broadly *portable*.

Subtlety: does not mean that *programs* need be portable!

- few applications C span full breadth
- many rely on implementation-defined behaviour
- *communication* needs knowledge of the *environment*

Keeping language portable avoids *incidental fragmentation*
“... the actual C language... has remained remarkably stable and unified compared to those of similarly widespread currency, for example Pascal and Fortran. ... On the whole, C has remained freer of proprietary extensions than other languages. ... [and has] succeeded in expressing programs, even including operating systems, on machines ranging from the smallest personal computers through the mightiest supercomputers.”
The spec doesn’t say “be unsafe”. It says... nothing!

- to protect *the specification’s* portability
- including to small / feeble / wacky devices
int *q = p + off; // if off is wrong, ...
*q = /* ... */; // ... anything could happen!

What *should* happen on a bounds error?
int *q = p + off; // if off is wrong, ...
*q = /* ... */;  // ... anything could happen!

What *should* happen on a bounds error?

- hardware exception?
int *q = p + off; // if off is wrong, ...
*q = /* ... */; // ... anything could happen!

What *should* happen on a bounds error?

- hardware exception?
- language-level exception?
int *q = p + off;   // if off is wrong, ...
*q = /* ... */;    // ... anything could happen!

What *should* happen on a bounds error?

- hardware exception?
- language-level exception?
- print a warning and carry on?
int *q = p + off; // if off is wrong, ...
*q = /* ... */; // ... anything could happen!

What *should* happen on a bounds error?

- hardware exception?
- language-level exception?
- print a warning and carry on?
- reboot machine?
`int *q = p + off; // if off is wrong, ...`

`*q = /* ... */; // ... anything could happen!`

What *should* happen on a bounds error?

- hardware exception?
- language-level exception?
- print a warning and carry on?
- reboot machine?

All are *plausible*.

Any specific choice would make the spec less portable.
Undefined behaviour is an inexorable consequence of:

1. A programming error may arise at run time

2. No particular method of handling it can be mandated

3. Can’t circumscribe possible dealings
Undefined behaviour is an inexorable consequence of:

1. A programming error may arise at run time  
   - total static checking is infeasible
2. No particular method of handling it can be mandated
3. Can’t *circumscribe* possible handlings
Undefined behaviour is an inexorable consequence of:

1. A programming error may arise at run time
   - total static checking is infeasible
2. No particular method of handling it can be mandated
   - ... that would be appropriate on all implementations
3. Can’t *circumscribe possible handlings*
Undefined behaviour is an inexorable consequence of:

1. A programming error may arise at run time
   - total static checking is infeasible

2. No particular method of handling it can be mandated
   - … that would be appropriate on all implementations

3. Can’t circumscribe possible dealings
   - … so can’t have impl-defined error behaviour
Undefined behaviour is an inexorable consequence of:

1. A programming error may arise at run time
   - total static checking is infeasible
2. No particular method of handling it can be mandated
   - ... that would be appropriate on all implementations
3. Can’t circumscribe possible handlings
   - ... so can’t have impl-defined error behaviour

If there is one platform where the arbitrary is excusable...
That’s a case in favour of undefined behaviour.

What’s the case against?
That’s a case in favour of undefined behaviour.

What’s the case against?

- it’s bad for debugging
- it’s bad for security, reliability, …

Is there really a tension here? (Hint: no.)
A well-known exponent of C:

“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?”

—Brian Kernighan
from The Elements of Programming Style (with P.J. Plauger)
Compilers have got very clever.

A tiny example due to Chris Lattner…

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

- line 2: compiler can assume P != 0
- line 5 is all that remains after optimisation
A well-known user of C who dislikes over-clever compilers:

“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 [cases allowing optimizations that exploit undefined behaviour] have ever shown up as issues.”
Implementation norms strike again.

- if execution would be undefined on $cond$
- ... assume $cond$ is false
- ... and transform code accordingly

While this is a permitted interpretation of “undefined”...
If compilers refrained from this, would things get too slow?

“Getting... [signed integer overflow] right... would generally force the compiler to add a few extra checks when you do... any kind of loop unrolling... and... 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.”

A more conservative response: we don’t know!

- TODO: build state-of-art non-UB-exploiting compiler
Picture credits:

- Ken and Den at work: unknown
- Dennis Ritchie (2011): Denise Panyik-Dale
- Brian Kernighan (2012): Ben Lowe
- Intel 8742: Ioan Sameli
- Lenovo laptop: Raimond Spekking
- Cray XMP48: Rama
- Linus Torvalds: Bret Hartman / TED
Big-picture recap:
- we say “language” but often mean “implementation”
- C gives first-class status to *communication*
- C impls need not be unsafe in practice
- undefined behaviour is a sane idea
- it’s not just about performance

Near-ish term research projects:
- build a dynamically safe C (atop...)
- specify address-space layout properties
- quantify how much gain comes from nasal demons

Thanks for listening! Questions?
Literature on adding dynamic checking to C is extensive.

- fat pointers
- referent objects / Jones & Kelly
- Memcheck / definedness checking
- SoftBound, CETS, …
- ASan, UBSan, …
- MPX, tagged memory, CHERI, …

However, norms hold sway there too.

- “adding checking to C” is the wrong viewpoint!
- considerable medium-level fruit remains
for (int i = 0; i < n; ++i)
{
    struct list_node *p = malloc(sizeof (struct list_node ));
    p->next = head;
    head = p;
}

for (int i = 0; i < m; ++i)
{
    unsigned out = 0;
    for (struct list_node *p = head; p; p = p->next)
    {
        out += p->x;
    }
    ret += out;
}

return ret;
uintptr_t = next_obj_loaded_above(stackptr)→l_addr;
for (const char **p_str = &environ[0]; *p_str; ++p_str)
{
    if ((uintptr_t) *p_str > lower
        && (uintptr_t) *p_str < upper)
    {
      /* It's pointing into the auxv's environ block.
         * Search for the adjoining auxv structures. */
      ElfW(auxv_t) *searchp
        = ((uintptr_t) *p_str) & ~(sizeof(void*) - 1);
      while (!(
             /* complex "found it" cond */ )))
      { searchp = /* next address to inspect */; } /* assert we found it */
      assert (searchp→a_type == AT_NULL
                && !searchp→a_un.a_val);
    }
}
If we want to check correctness of code like that...  

- need knowledge of *how different areas of memory relate*
- where? higher, lower, within $n$ bytes?
- what properties hold of the data encodings?

A bit like run-time type information...  

- not “for C”; for the whole process!

... also covering large-scale *layout* properties  

- what is inhabiting my address space, and where?

A research paper might just appear later on this...