Patterns Of Human Error

by Walter Bright

digitalmars.com
“It can only be attributable to human error.”

– Hal 9000
Eliminating Human Error

- Look at lists of found errors
- See if there is a pattern to them
- Redesign so that pattern becomes impossible
  - Human error is out of the loop
  - No need to train to not make those mistakes
  - No need for QA to look for those mistakes
  - The AE35 unit would never have failed

“A clever person solves a problem. A wise person avoids it.” --Einstein
4 Bolt Mounting Pattern
Offset Hole
Principles

- Easy to get it right
- Hard to get it wrong
- When it looks right, it is right
- When it looks wrong, it is wrong
- Symmetry is not always the best design
Typical Car Battery
A Sea of Wrong

- Battery can be installed in two orientations
- Cables are long enough to connect to either post
- Cable connectors fit on either post
- Cables are not marked, not color coded
- Battery + and – markings are hard to see
Does This Matter?

Let's find out!
"Long story short, my OEM battery installs one way and my aftermarket installs the opposite direction where you would have to turn it around in order to install the aftermarket. Well, you can see where this is going to lead too. I went back to an OEM battery and didn't even look at the positive/negative signs. I just installed it the same way the aftermarket sat in there....I saw a few sparks and that threw a red flag so I checked it and realized that it was reversed. Installed it the right way and now I have no power at all. Nothing lights up."

Better Battery Design

How could this be improved on?
AV Rule 14: *Literal suffixes shall use uppercase rather than lowercase letters.*

```c++
const int64 fs_frame_rate = 64l; // Wrong! Looks too much like 641
const int64 fs_frame_rate = 64L; // Okay
```


Simple fix: make l suffix illegal. No more possibility of this error. End of story.
I Know What You're Thinking, Punk

Was that 5 zeros, or 6?

int i = 1000000;

Do you feel lucky?
We Doan Need No Steenkin' Luck

```java
int i = 1_000_000;
```

Code that looks right should be right.

This particular improvement, though trivial, has been a much liked and appreciated one.
What Should This Do?
What Does It Do?

\[ a < b < c \]
Most Would Expect

\[ a < b \land b < c \]

And that would happen in Python, but not in C. In C it is:

\[ ((a < b) ? 1 : 0) < c \]

Which pretty much nobody wants
Ways To Fix It

• Make it work like Python
  • But that would be a silent, unexpected result for the off chance that someone actually intended the C behavior
    − Things like this make users very mad
• Run a third party static analysis tool
  • Bah
• Fix the grammar so a<b<c won't even compile
  • +1
The Old Grammar

CmpExpression:
  ShiftExpression
  CmpExpression < ShiftExpression
  CmpExpression > ShiftExpression
  CmpExpression <= ShiftExpression
  CmpExpression >= ShiftExpression
Fixed Grammar

CmpExpression:
  ShiftExpression
  ShiftExpression < ShiftExpression
  ShiftExpression > ShiftExpression
  ShiftExpression <= ShiftExpression
  ShiftExpression <= ShiftExpression
Post Mortem

- Works
- Simple & clean to implement
- Compatible with C
  - No silently different results
- Can still use (a<b)<c
  - Which looks intentional
- No complaints from the field
Another Grammar Issue

\[ a \& b < c \]

Which means:

\[ a \& (b < c) \]

What was meant was:

\[ a \&\& (b < c) \]

Or:

\[ (a \& b) < c \]
AndExpression:
  CmpExpression
  AndExpression & CmpExpression
Fixed Grammar

AndAndExpression:
  OrExpression
  AndAndExpression && OrExpression
  CmpExpression
  AndAndExpression && CmpExpression

AndExpression:
  ShiftExpression
  AndExpression & ShiftExpression
Enough Tiddlywink Issues

- Let's do something real
- How about C buffer overflows?
C's Biggest Mistake

- No, not operator precedence
- Not null pointers
- Arrays are converted to pointers when passed to a function
  - Thereby losing information about the array length

```c
void foo(T *array);
...
T array[6];
...
foo(array);
```
Typical Solution

void foo(size_t dim, T* array);
... 
foo(sizeof(array)/sizeof(T), array);

Obviously inadequate or we wouldn't be expending vast sums finding and fixing problems with it.
CERT – C Secure Coding Standard

- Some guidelines to follow
  - List of Do's and Don'ts
- More based on hope rather than guarantee

https://www.securecoding.cert.org/confluence/display/seccode/06.+Arrays+%28ARR%29
Microsoft SAL Solution

```c
void foo(size_t dim, __in_bcount_full(dim) T* array);
...
foo(sizeof(array)/sizeof(T), array);
```
D Solution – Phat Pointers

aka *dynamic arrays*, consisting of a pointer paired with a length:

```c
void foo(T[] array);
...
foo(array);
```

http://drdobbs.com/blogs/architecture-and-design/228701625
An Innocuous C Function

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

typedef long T;

bool find(T *array, size_t dim, T t) {
    int i;
    for (i = 0; i <= dim; i++) {
        int v = array[i];
        if (v == t)
            return true;
    }
    return false;
}
```
Common Error Patterns

- i should be size_t
- <= should be <
- extraneous ;
- v should be type T
- missing return

But it still compiles!
Error Patterns Eliminated

- loop index is inferred
- loop termination is inferred
- ; is not allowed as a loop body
- type of v is inferred
- falloff with no return value not allowed

```c
alias long T;

bool find(T[] array, T t) {
    foreach (v; array) {
        if (v == t) {
            return true;
        }
    }
    return false;
}
```
Error Codes

- Ignored by default
- Be sure and clean up along all paths
  - These are rarely 100% tested, not even close
local int gz_init(gz_statep state) {
    int ret;
    z_streamp strm = &(state->strm);
    /* allocate input and output buffers */
    state->in = malloc(state->want);
    state->out = malloc(state->want);
    if (state->in == NULL || state->out == NULL) {
        if (state->out != NULL) free(state->out);
        if (state->in != NULL) free(state->in);
        gz_error(state, Z_MEM_ERROR, "out of memory");
        return -1;
    }
    /* allocate deflate memory, set up for gzip compression */
    strm->zalloc = Z_NULL;
    strm->zfree = Z_NULL;
    strm->opaque = Z_NULL;
    ret = deflateInit2(strm, state->level, Z_DEFLATED, 15+16, 8, state->strategy);
    if (ret != Z_OK) {
        free(state->in);
        gz_error(state, Z_MEM_ERROR, "out of memory");
        return -1;
    }
    state->size = state->want; /* mark state as initialized */
    /* initialize write buffer */
    strm->avail_out = state->size;
    strm->next_out = state->out;
    state->next = strm->next_out;
    return 0;
}
local int gz_init(gz_statep state) {
    int ret;
    z_streamp strm = &(state->strm);
    /* allocate input and output buffers */
    state->in = malloc(state->want);
    state->out = malloc(state->want);
    if (state->in == NULL || state->out == NULL) {
        if (state->out != NULL) free(state->out);
        if (state->in != NULL)   free(state->in);
        gz_error(state, Z_MEM_ERROR, "out of memory");
        return -1;
    }
    /* allocate deflate memory, set up for gzip compression */
    strm->zalloc = Z_NULL;
    strm->zfree = Z_NULL;
    strm->opaque = Z_NULL;
    ret = deflateInit2(strm, state->level, Z_DEFLATED, 15+16, 8, state->strategy);
    if (ret != Z_OK) {
        free(state->in); /* BUG: what about state->out ? */
        gz_error(state, Z_MEM_ERROR, "out of memory");
        return -1;
    }
    state->size = state->want; /* mark state as initialized */
    /* initialize write buffer */
    strm->avail_out = state->size;
    strm->next_out = state->out;
    state->next = strm->next_out;
    return 0;
}
Exception Handling

- Not ignored by default
- Even decent error messages by default
- Cleanup & recovery code tends to be simpler
local int gz_init(gz_statep state) {
    z_streamp strm = &(state->strm);

    /* allocate input and output buffers */
    state->in = malloc(state->want);
    scope (failure) free(state->in);
    state->out = malloc(state->want);
    scope (failure) free(state->out);

    /* allocate deflate memory, set up for gzip compression */
    strm->zalloc = Z_NULL;
    strm->zfree = Z_NULL;
    strm->opaque = Z_NULL;
    deflateInit2(strm, state->level, Z_DEFLATED,15+16, 8, state->strategy);

    state->size = state->want;  /* mark state as initialized */

    /* initialize write buffer */
    strm->avail_out = state->size;
    strm->next_out = state->out;
    state->next = strm->next_out;
    return 0;
}
Stupid Pointer Bugs

• Endless pain
• Caused by:
  • Uninitialized memory
  • Out of bounds pointer arithmetic
  • Mismatched malloc / free
  • Breaking the type system (casts & unions)
Memory Safety Guarantees

- Java proves it's possible to design semantics so that pointer bugs are impossible
  - By eliminating those 4 features that are the source of pointer bugs
- But that imposes some severe limitations
  - Which pushes programmers back to unsafe languages like C
- We do want those apples
  - but don't like little green worms
Separate Code into Safe and Unsafe

- Done at function level by marking functions or groups of functions as
  - @safe
  - @system
  - @trusted
- Default is @system
Safe Functions

- Are guaranteed to be memory safe
  - i.e. no pointer bugs

- Not allowed:
  - Calling system functions
  - Pointer arithmetic
  - Unsafe casts or unions
  - Uninitialized memory
System Functions

- Can do anything
- For example, C's free(void*) is a system function
- But onus is on programmer to do it right
Trusted Functions

• Form a bridge between safe and system functions
• Safe functions can call trusted functions
• Trusted functions provide a safe interface to system functions
• Guaranteed by the programmer of the trusted function, not the language
Not A Magic Bullet, but...

- Greatly reduces the scope of any stupid pointer bugs
- The vast bulk of an app should be safe code
  - And hence mechanically guaranteed
- System code should only be a small part
  - Making it easier to manually verify correctness
- Far better than your entire program being uncheckable system code
Other Bug Prevention Measures

- Checkable function purity
- Immutable data structures that are “turtles all the way down”
- Function anti-hijacking
- Default to thread local data, shared data must be explicitly typed as shared
- Value range propagation
Coming Soon

- User defined data subtyping
  - Classic example: NonNull!(T)
  - Ranged numeric types: Ranged!(int,0,10)
  - Validated data: Validated!(T)
  - Takes place of Hungarian Notation
    - http://www.joelonsoftware.com/articles/Wrong.html
Conclusion – eliminate stupid bugs

- Unreliable (subject to human error)
  - Relying on luck
  - Coding standards
  - Code review
  - Better education
  - Hire better programmers

- Reliable
  - Make errors impossible
  - Make errors hard to write
  - Make correct code easy to write
What Patterns Of Error Affect You?