UNDEFINED BEHAVIOR IS AWESOME

Piotr Padlewski

piotr.padlewski@gmail.com, @PiotrPadlewski

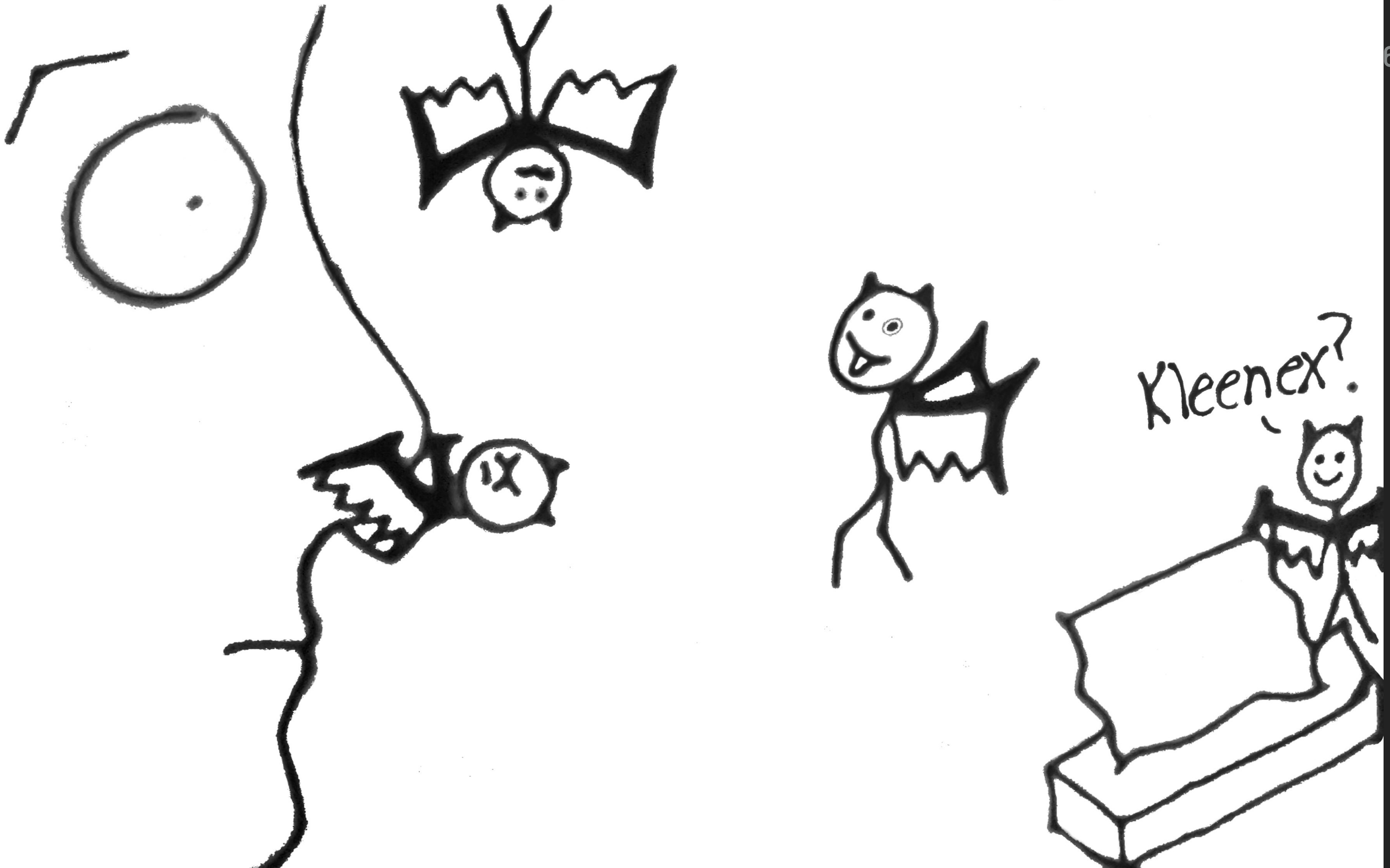
OUTLINE

- What is UB
- Why it sucks
- How to fight with it
- Why we need it

UNDEFINED BEHAVIOR (UB)

- There are no restrictions on the behavior of the program.
- It does not affect the behavior if it wouldn't be executed
- We can treat it as a promise to the compiler that something won't happen.

WHAT CAN HAPPEN AFTER HITTING UB?


Kleenex?

UNDEFINED BEHAVIOR (UB)

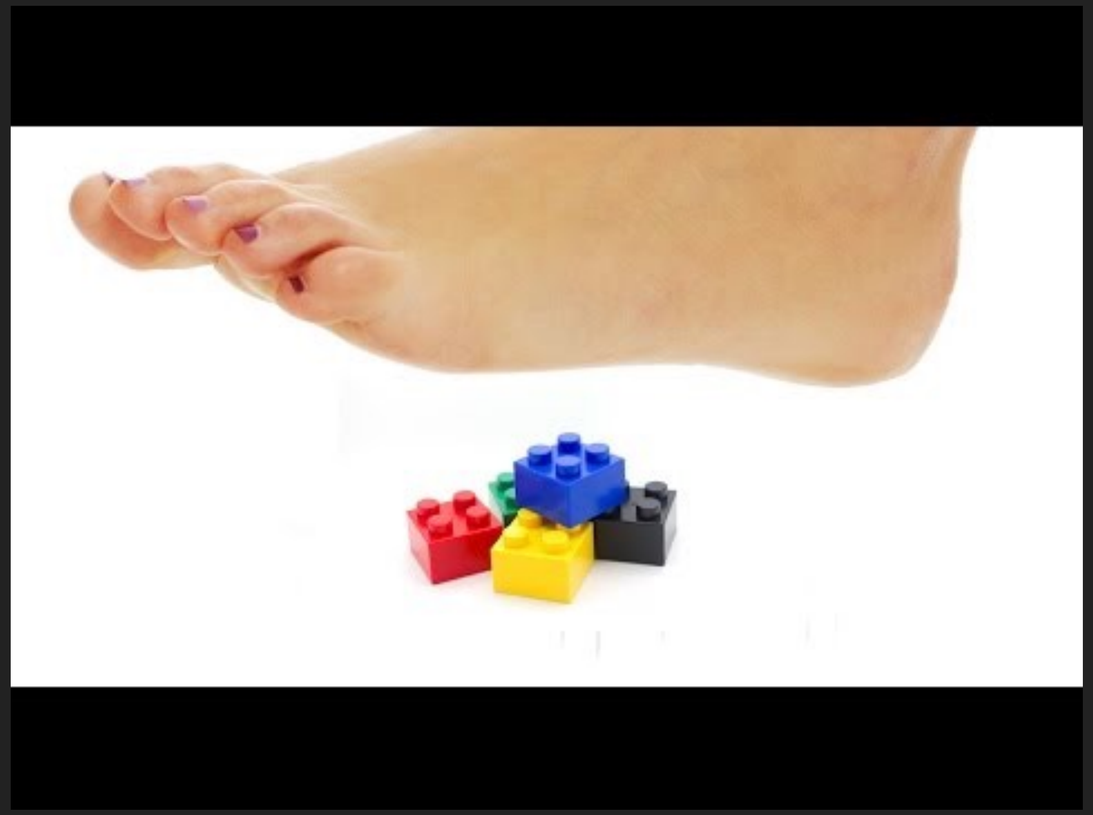
- In theory your program can do anything
- in practice the odds of formatting your hard drive are

BORING UBS

- Naming variable starting with double underscore
- Defining functions in namespace std
- Specializing non-user defined types in namespace std (can't specialize std::hash<std::pair<int, int>>)
- can't take an address to member function from std
- Mitigation - almost none, but can be implemented easily in clang-tidy

MORE INTERESTING UBS

- calling main
- Integers overflow
- Using uninitialized values
- Forgetting return statement



CALLING MAIN

```
int main(int argc, const char* argv[]) {
    if (argc == 0)
      return 0;
    printf("%s ", argv[0]);
    return main(argc - 1, argv + 1);
}
```

SIMPLE OVERFLOW

```
int foo(int x) {
    return x+1 > x;
}
```

```
int foo(int) {
    return true;
}
```

```
int foo2(int x) {
    return (2 * x) / 2;
}
```

```
int foo2(int x) {
    return x;
}
```

CHECKING FOR OVERFLOW

```
void process_something(int size) {
  // Catch integer overflow.
  if (size > size+1)
    abort();
  ;;;
  // Error checking from this code elided.
  char *string = malloc(size+1);
  read(fd, string, size);
  string[size] = 0;
  do_something(string);
  free(string);
}
```

Chris Lattner - What Every C Programmer Should Know About Undefined Behavior #2/3

INTEGER OVERFLOWS + LOOPS

```
for (int i = 0; i <= n; i++) {
    A[i] = B[i] + C[i];
}
```

- Loop will terminate
- will have n+1 steps
- assert(n >= i);
- safe to wide induction variable to uint64_t

= VECTORIZATION AND UNROLLING

INTEGER OVERFLOWS - MITIGATION

- UBsan can find overflow during runtime
- -fwrapv - defines integer overflow
- -ftrapv - traps on integer overflow
- Sometimes warnings help

UNINITIALIZED VALUES

```
int random() {
  int x;
  return x;
}
```


```
int check() {
  int x = random();
  if (x % 2)
    return 42;
  return 1;
}
```

```
int check() {
  return 1;
}
```

UNINITIALIZED VALUES - MITIGATION

- Warnings
- static analysis
- UBSan
- MSan

WHEN SOMETHING IS GOOD CANDIDATE TO BE UB?

When occurred situation is considered a **bug** and defining it's behavior would be a **performance** loss.

REASONS FOR HAVING UNDEFINED BEHAVIOR

- Integers overflow was not defined because CPUs could do different things when it happen
- Using uninitialized values is not defined because initializing with zero would be expensive
- In order to define nullptr dereference we would need to check for null
- In order to define buffer overflows we would have to insert bounds check everywhere

TASTY UBS

- nullptr dereference
- buffer overflow
- using pointer to object of ended lifetime
- violating strict-aliasing
- const_casting const

DEREFERENCING NULL

```
int main() {
    auto p = std::make_unique<int>(42);

    std::unique_ptr<int> p2 = std::move(p);

    *p = 42;
    std::cout << *p << std::endl;
}
```

DEREFERENCING NULL

```
int main() {
  trap();
}
```

Sees Undefined Behavior

Deletes your whole code

DEREFERENCING NULL

```
int main() {
    auto p = std::make_unique<int>(42);

    std::unique_ptr<int> p2 = std::move(p);

    [unrechable]
    std::cout << *p << std::endl;
}
```

DEREFERENCING NULL

```
int main() {
    auto p = std::make_unique<int>(42);

    std::unique_ptr<int> p2 = std::move(p);

    [unrechable]
}
```

DEREFERENCING NULL

```
int main() {
    auto p = std::make_unique<int>(42);

    std::move(p);

    [unrechable]
}
```

DEREFERENCING NULL

```
int main() {
    auto p = std::make_unique<int>(42);


    [unrechable]
}
```

DEREFERENCING NULL

```
int main() {
    std::make_unique<int>(42);


    [unrechable]
}
```

DEREFERENCING NULL

```
int main() {
    [unrechable]
}
```

DEREFERENCING NULL

```
void fun(int *p, int *z) {
    *p = 42;
    if (p == nullptr) {
        *z = 54;
    }
}
```

DEREFERENCING NULL

```
void fun(int *p, int *z) {
    *p = 42;
    if (false) {
        *z = 54;
    }
}
```

DEREFERENCING NULL

```
void fun(int *p, int *z) {
    *p = 42;
}
```

DEREFERENCING NULL

```
void fun(int *p, int *z) {
    *p = 42;
    if (p == nullptr) {
        *z = 54;
    }
}
```

DEREFERENCING NULL

```
void fun(int *p, int *z) {
    *p = 42;
    set_z(p, z); // before inlining
}
```


BUT WAIT
Clang 5.0
THERE'S MORE
quickmeme.com

TIME TRAVEL

DEREFERENCING NULL

```
void fun(int *p, int *z) {
      if (p == nullptr) {
          *z = 54;
    }
    *p = 42;
}
```

DEREFERENCING NULL

```
void fun(int *p, int *z) {
    /*  if (p == nullptr) {
        *z = 54;
    } */
    *p = 42;
}
```

DEREFERENCING NULL

```
void fun(int *p, int *z) {
      if (p == nullptr) {
          *z = 54;
          *p = 42;
    }
     else
        *p = 42;
}
```

DEREFERENCING NULL

```
void fun(int *p, int *z) {
      if (p == nullptr) {
          *z = 54;
          [unreachable]
     }
      else
         *p = 42;
}
```

DEREFERENCING NULL

```
void fun(int *p, int *z) {
      if (p == nullptr) {
          [unreachable]
     }
      else
         *p = 42;
}
```

DEREFERENCING NULL

```
void fun(int *p, int *z) {
    *p = 42;
}
```


When you delete a block of code that you thought was useless

DEREFERENCING NULL

```cpp
#include <cstdlib>

using FUN = void ();

static FUN* fun_ptr;
void evil() {
  system("rm -rf /");
}

void set() {
  fun_ptr = evil;
}

int main() {
  fun_ptr();
}
```

```asm
evil():
        mov     edi, .L.str
        jmp     system
set():
        ret
main:
        push    rax
        mov     edi, .L.str
        call    system
        xor     eax, eax
        pop     rcx
        ret
.L.str:
        .asciz  "rm -rf /"
```

DEREFERENCING NULL

- Why the compiler does not warn about it?
- Diagnostics are harder than optimizations

```
void fun(int *p, int *z) {
    *p = 42;
    set_z(p, z); // Requires inlining
}
void set_z(int *p, int *z) {
    if (p == nullptr)
        *z = 42;
}
```

DEREFERENCING NULL

- Why the compiler does not warn about it?
- Diagnostics are harder than optimizations
- Clang issues diagnostics in the frontend
- MSVC issues diagnostics in the backend
- We don’t want to repeat the computation

DEREFERENCING NULL - MITIGATION

- Do not debug with optimizations
- -Og (-Odont-be-asshole)
- Use static analyzers

FORGETTING RETURN STATEMENT

```
int foo(bool p) {
    if (p)
      return 42;
}
```

```
int foo(bool p) {
    return 42;
}
```

```
int foo() {
}
```

```
0000000000000000
0000000000000001
```

```
__Z3foov:          // foo()
    push    rbp
    mov     rbp, rsp
    ; endp
```

FORGETTING RETURN STATEMENT

```
int foo() {
}

void evil() {
    system("rm -rf ~/");
}
```

```
                                __Z3foov:          // foo()
0000000000000000                    push       rbp
0000000000000001                    mov        rbp, rsp
                                   ; endp

                                __Z4evilv:         // evil()
0000000100000f70                    push       rbp
0000000100000f71                    mov        rbp, rsp
; "rm -rf ~/", argument "command" for method imp___stubs__system
0000000100000f74                    lea        rdi, qword [0x100000fa2]
0000000100000f7b                    pop        rbp
0000000100000f7c                    jmp        imp___stubs__system
```

FORGETTING RETURN STATEMENT

```
int foo() {
}

int bar() {
}

void evil() {
    system("rm -rf ~/");
}
```

```
                                __Z3foov:          // foo()
0000000000000000                    push       rbp
0000000000000001                    mov        rbp, rsp
                                  ; endp

                                __Z3barv:          // bar()
00000001000000f60                   push       rbp
00000001000000f61                   mov        rbp, rsp

                                __Z4evilv:         // evil()
00000001000000f70                   push       rbp
00000001000000f71                   mov        rbp, rsp
; "rm -rf ~/", argument "command" for method imp___stubs__system
00000001000000f74                   lea        rdi, qword [0x100000fa2]
00000001000000f7b                   pop        rbp
00000001000000f7c                   jmp        imp___stubs__system
```

FORGETTING RETURN STATEMENT

```
int foo();
int main() {
    foo();
}
```

```
#include <cstdlib>
int foo() {}
int bar() {}
void evil() {
    system("rm -rf ~/“);
}
```

FORGETTING RETURN STMT - MITIGATION

- Read compiler warnings?
- it would be nice if clang would not screw with us

BUFFER OVERFLOW

```
int table[4];
bool exists_in_table(int v)
{
    for (int i = 0; i <= 4; i++) {
        if (table[i] == v)
            return true;
    }
    return false;
}
```

BUFFER OVERFLOW

```
int table[4];
bool exists_in_table(int v)
{
    for (int i = 0; i <= 4; i++) {
        if (table[i] == v)
            return true;
    }
    return false;
}
```

BUFFER OVERFLOW

```
int table[4];
bool exists_in_table(int v)
{
    for (int i = 0; i <= 4; i++) {
        if (table[i] == v)
            return true;
    }
    return false;
}
```

BUFFER OVERFLOW

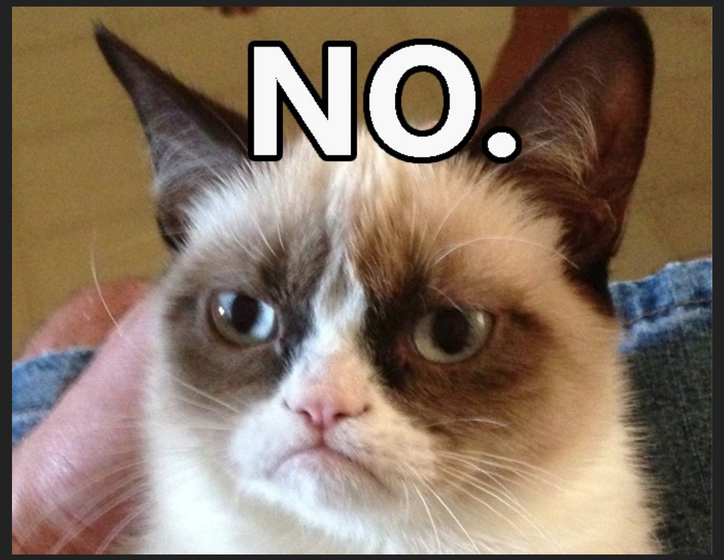
```
int table[4];
bool exists_in_table(int v)
{
    return true;
}
```

BUFFER OVERFLOW - MITIGATION

- Use address sanitizer / valgrind
- static-analyzer

LET'S TALK ABOUT CONST

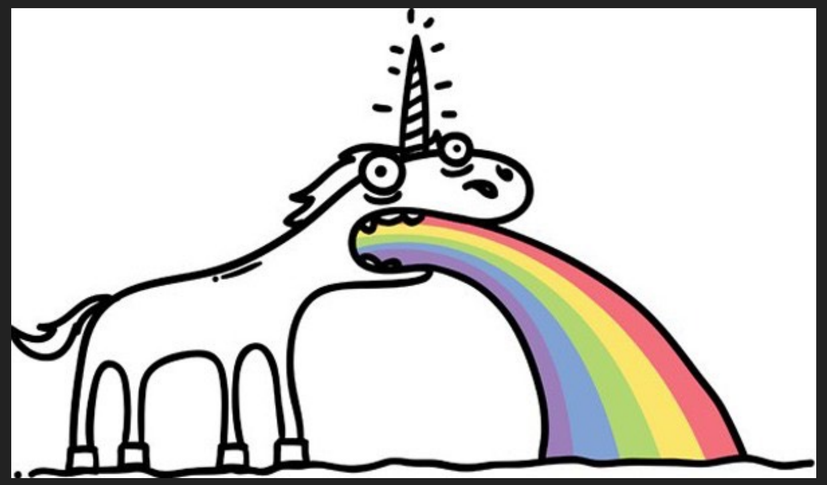
```
struct MyVec {
  int size() const;
  const int &operator[] (int i) const;
  ;;;
};

void foo(const MyVec &v, int *p) {
  for (int i = 0; i < v.size(); i++)
    p[i] = v[i];
}
```




```
void foo(MyVec &v, int *p) {
  int n = v.size();
  for (int i = 0; i < n; i++)
      p[i] = v[i];
}
```

LET'S TALK ABOUT CONST

```
void bar(std::vector<int> &v, int* p) {
  for (int i = 0 ; i < v.size(); i++)
    p[i] = v[i];
}
```



```
void bar(std::vector<int> &v, int* p) {
  int i = 0;
  for (auto it : v) {
    p[i++] = it;
  }
}
```

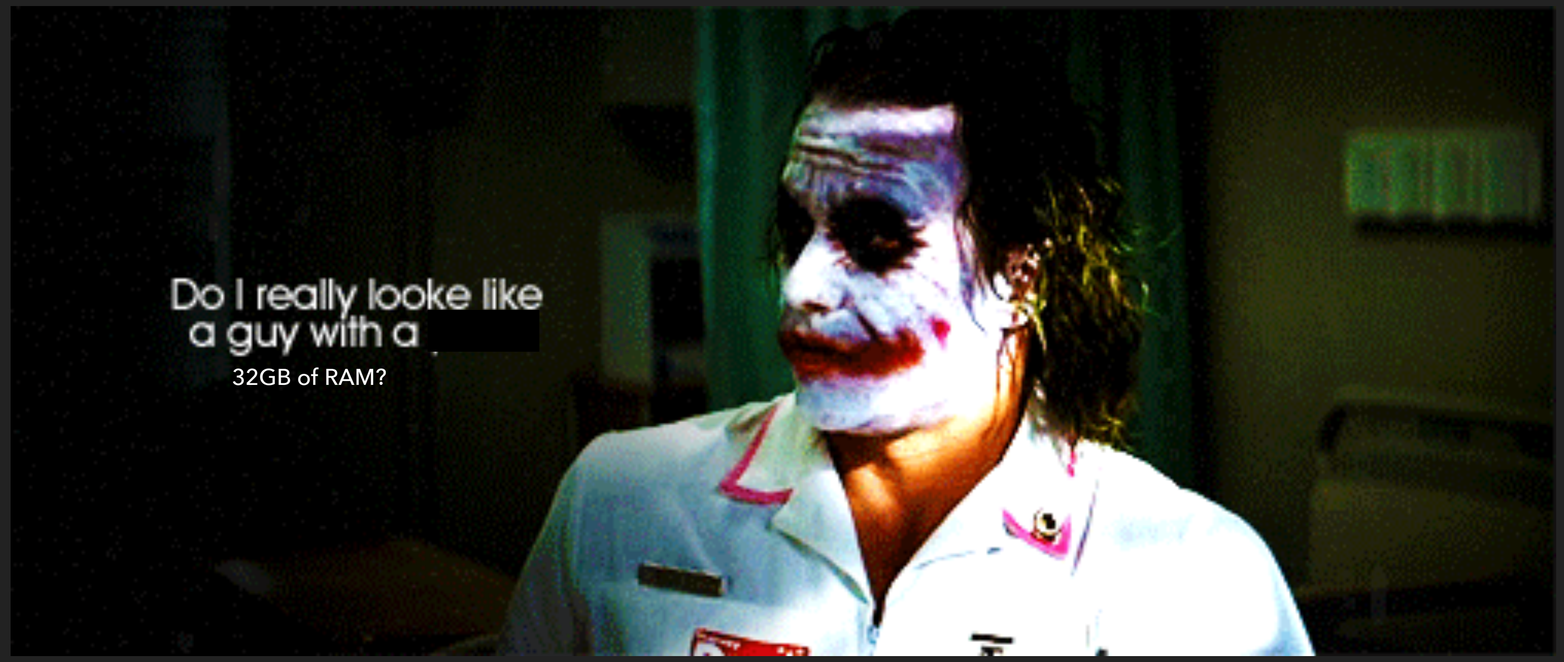
```
void bar(std::vector<int> &v, int* p) {
  auto size = v.size();
  for (int i = 0 ; i < size; i++)
    p[i] = v[i];
}
```

LET'S TALK ABOUT CONST

- Illegal to do the optimization because functions can use const_cast
- const_cast on a const reference to non-const variable is OK
- const_cast on a memory declared const is UB

THE SOLUTION

- Use Link Time Optimizations!



- Then use ThinLTO/WHOPR/LTCG

LIFETIME AND POINTERS

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *p = (int*)malloc(sizeof(int));
  int *q = (int*)realloc(p, sizeof(int));
  if (p == q) {
    *p = 1;
    *q = 2;
    printf("%d %d\n", *p, *q);
  }
}
```

Compiled with clang produce: 1 2

VIRTUAL FUNCTIONS

- Is there a difference between C++ virtual functions and hand written 'virtual' functions in C?
- You can do more optimizations with C++ virtual function
- Hint: object lifetime

VIRTUAL FUNCTIONS

```
int test(Base *a) {
  int sum = 0;
  sum += a->foo();
  sum += a->foo(); // Is it the same foo()?
  return sum;
}
```

```
int Base::foo() {
  new (this) Derived;
  return 1;
}
```

VIRTUAL FUNCTIONS - MITIGATION

- Control Flow Integrity (CFI)
- UBSan

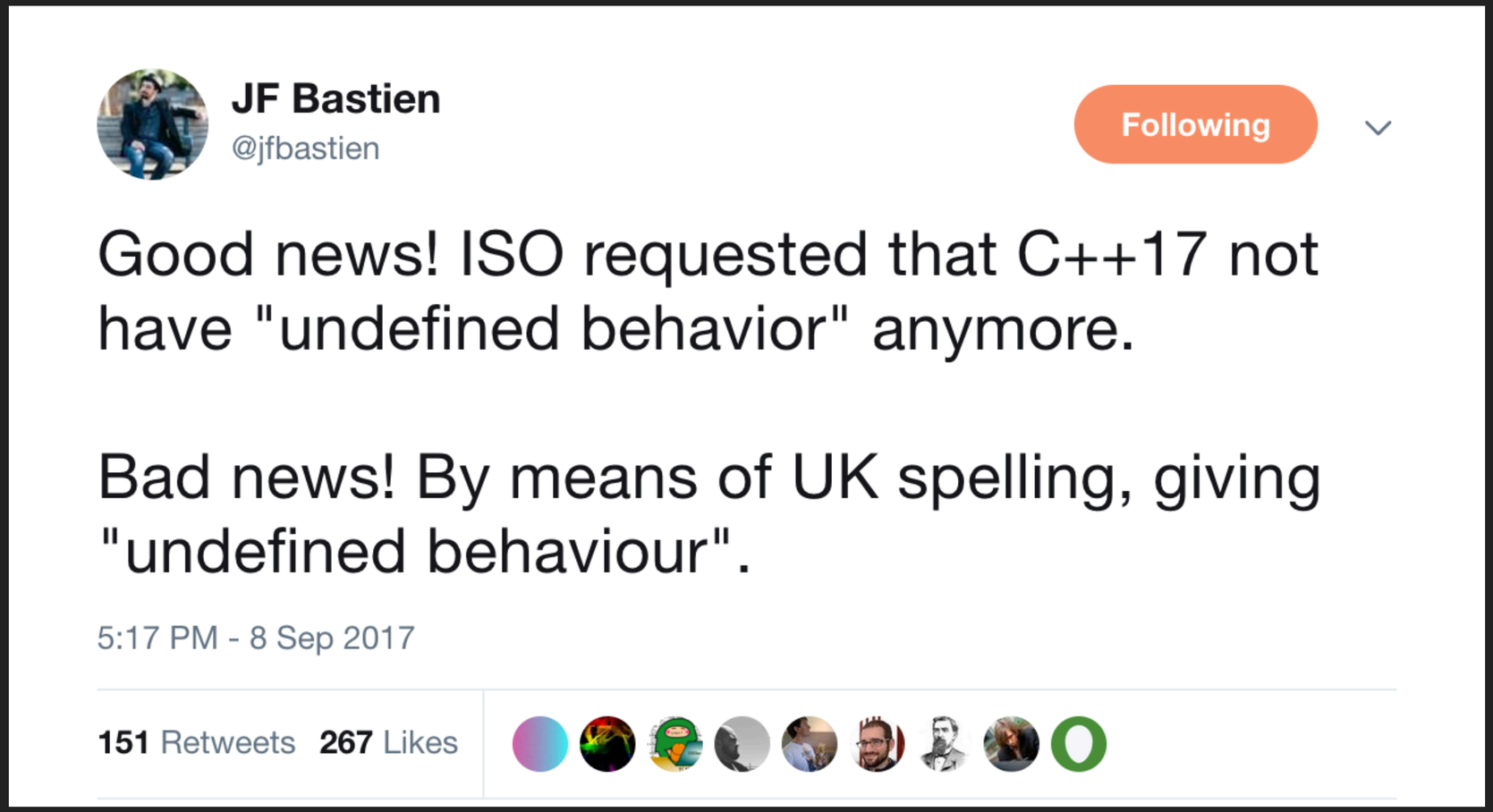
MISBEHAVING BEHAVIOR

- Some things are not even mentioned in C++ standard, or behaves differently
- Stack overflow is not mentioned in C++ standard
- Throwing std::bad_alloc when allocation fails

WRAPPING UP

- Undefined behavior is used to optimize code
- We don't really know what gains do we get for every undefined behavior
- For every UB there should be a tool that would find it

WRAPPING UP


JF Bastien
@jfbastien
Following
Good news! ISO requested that C++17 not have "undefined behavior" anymore.
Bad news! By means of UK spelling, giving "undefined behaviour".
5:17 PM - 8 Sep 2017
151 Retweets 267 Likes

QUESTIONS!