UNDEFINED BEHAVIOR IS AWESOME

Piotr Padlewski

piotr.padlewski@gmail.com, @PiotrPadlewski

ABOUT MYSELF

- Currently working in IIIT developing C++ tooling like clang-tidy and studying on University of Warsaw.
- Worked on optimizations in clang and LLVM - devirtualization and ThinLTO at Google

IMPLEMENTATION DEFINED BEHAVIOUR (IB)

- The behavior of the program varies between implementations, and the conforming implementation must **document** the effects of each behavior.
- Examples:
- Type of std::size_t
- Number of bits in long
- Number of bits in a byte

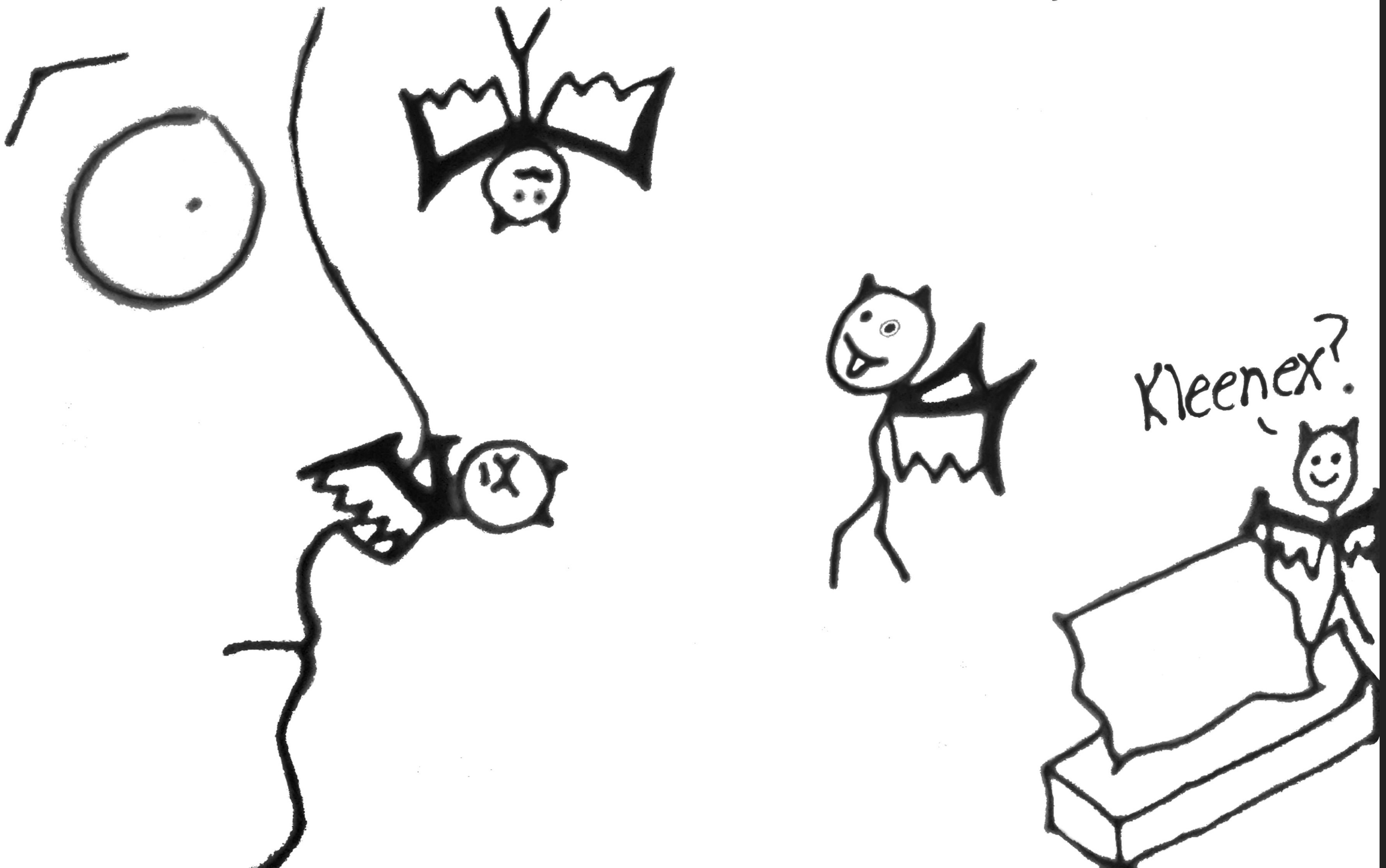
UNSPECIFIED BEHAVIOR

- The behavior of the program varies between implementations and the conforming implementation is not required to document the effects of each behavior.[0]
- Examples:
- Order of evaluation: foo(bar(), baz());
- Identical string literals address: “foo” == “foo”

UNDEFINED BEHAVIOR (UB)

- There are no restrictions on the behavior of the program.
- 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 underscore
- Defining functions in namespace std
- Specializing non-user defined types in namespace std (can’t specialize std::hash<std::pair<int, int>>)
- Calling delete/free after new[]

MORE INTERESTING UBS

- Dereferencing nullptr
- Using uninitialized values
- Integers overflows

SIMPLE OVERFLOW

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

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

LOOPS

```
for (int i = 0; i < n; i+=2) {
    A[i] = B[i] + C[i];
    A[i+1] = B[i+1] + C[i+1];
}
```

- Loop will terminate
- assert(n >= i);
- safe to wider i to uint64_t

= VECTORIZATION AND UNROLLING

TASTY UBS

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

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;
}
```

LIFETIME AND POINTERS

```c
#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

LIFETIME AND POINTERS

```
vector<int> v;
v.reserve(2);
v.push_back(4);
v.push_back(2);
auto& ref = v[0];
v.push_back(42);

if (&v[0] == &ref)
    ref = 42;
```

IS THIS VALID?
IS THIS UB/IB?

valid, because std::allocator
only calls new/delete

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.

STACK OVERFLOW

- Why I can’t get a nice error message saying I got stack overflow?

OUT OF MEMORY

- Ok, at least we get std::bad_alloc or nullptr when heap allocation fails.

WHY PEOPLE HATE EXCEPTIONS

- With enabled exceptions **every** call generate branch
- Compiling with -fno-exceptions changes every throw to call of std::abort()
- Other solution is to mark almost every function with noexcept

LET'S TALK ABOUT CONST

```
struct A {
  void foo() const;
  int b = 0;
};

void bar(int);

int main() {
   A a;
   bar(a.b);
   a.foo();
   bar(a.b):
}
```

```
struct A {
  void foo() const;
  int b = 0;
};

void bar(int);

int main() {
   A a;
   bar(0);
   a.foo();
   bar(0):
}
```

LET'S TALK ABOUT CONST

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

CONST PROPAGATION

- Every time we call external const method, or function having const reference parameters, the compiler have to assume the worst - const_cast
- This really sucks, because const propagation is awesome
- If only const would have non-mutable guarantee…

IMAGINE THERE IS NO CONST_CAST

CONST PROPAGATION WITH STRICT CONST

```
void foo(const int &a) {
    bar(a);
    bar(a);
}

void bar(const int &b);
```

CONST PROPAGATION WITH STRICT CONST

```
void foo(const int &a) {
    const int temp = a;
    bar(temp);
    bar(temp);
}

void bar(const int &b);
```

```
int global;

void caller() {
    foo(global);
}

void bar(const int &b) {
    global++;
}
```

WHAT ABOUT MUTABLE?

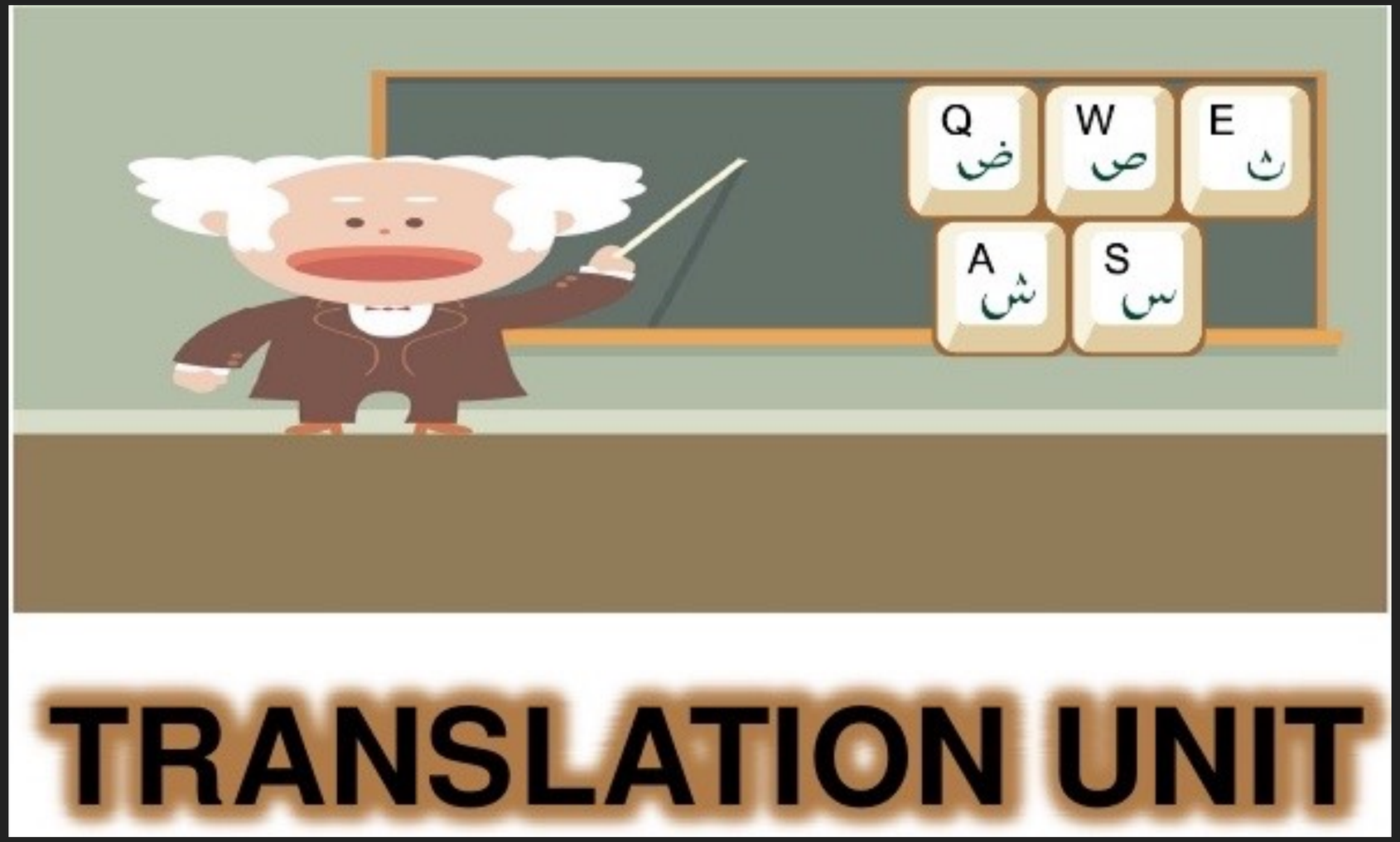
```
class A {
    ;;;
};

void caller() {
    A a;
    foo(a);
}

void foo(const A &a);
```

```
class A {
    mutable X;
};

void caller() {
    A a;
    foo(a);
}

void foo(const A &a);
```

DOES THE REAL CONST EXIST?

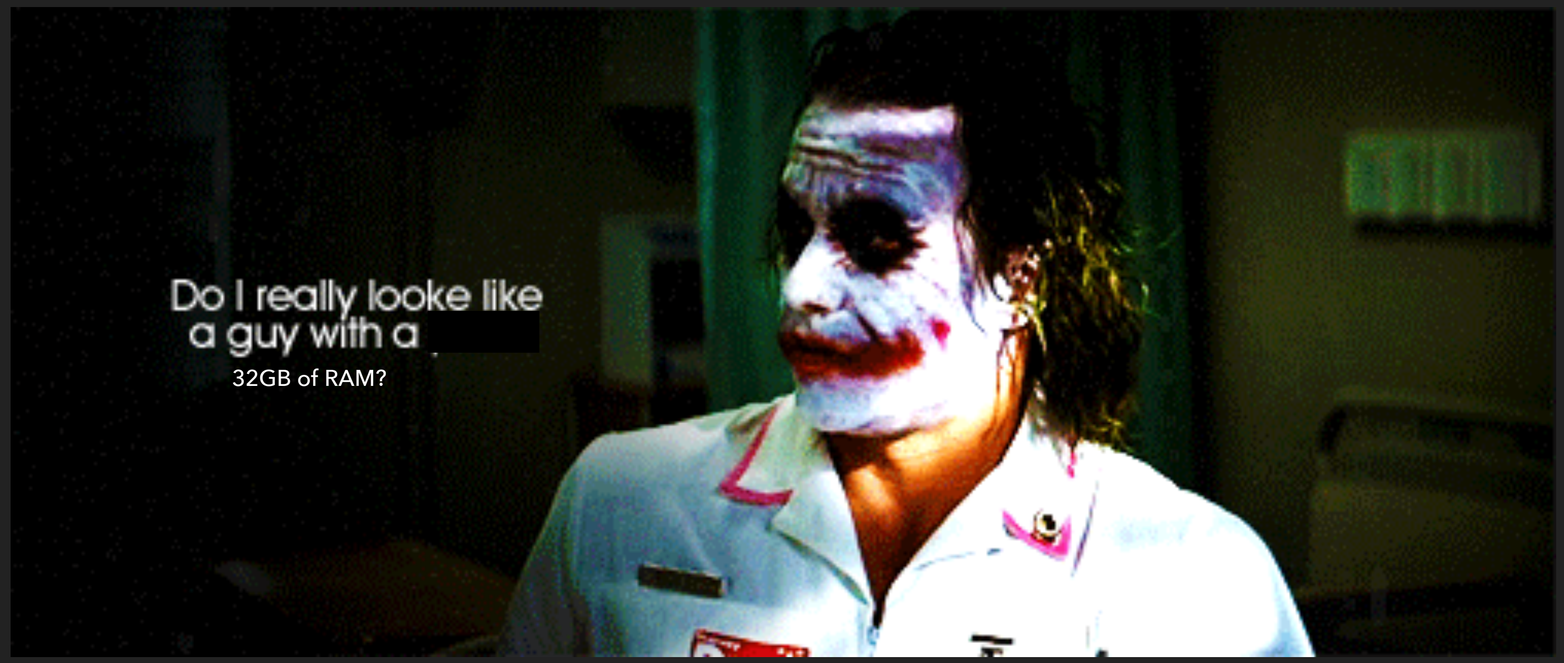
- Kinda. Compilers mark functions as „const“ (e.g. readonly in llvm) if they don’t modify memory
- Maybe problem is somewhere else?

BUT IS THE CONST THE REAL PROBLEM?



THE SOLUTION

- Use Link Time Optimizations!



- Then use ThinLTO/LIPO

VIRTUAL FUNCTIONS

- Is there a difference between C++ virtual functions and hand written 'virtual' functions in C?
- C++ standard doesn’t explicitly mention UB with virtual functions
- But it say much about 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;
}
```

CALLING MAIN

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

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

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

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

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

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

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

    std::move(p);
}
```

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

}
```

```
int main() {


}
```

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

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

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

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

QUESTIONS!