Undefined Behavior in 2017

John Regehr
University of Utah, USA

Today:

- What is undefined behavior (UB)?
- Why does it exist?
- What are the consequences of UB in C and C++?
- Modern UB detection and mitigation

sqrt(-1) = ?

- i
- NaN
- An arbitrary value
- Throw an exception
- Abort the program
- Undefined behavior

- **Undefined behavior** (UB) is a design choice
- UB is the most efficient alternative because it imposes the fewest requirements
  - "… behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements"

C and C++ have lots of UB

- To avoid overhead
- To avoid compiler complexity
- To provide maximal compatibility across implementations and targets

According to Appendix J of the standard, C11 has 199 undefined behaviors

- But this list isn't complete
- And there's no comparable list for C++
- And new UBs are being added

8) Integer, floating-point, or enumeration type can be converted to any complete enumeration type. The result is unspecified (until C++17) undefined behavior (since C++17) if the value of *expression*, converted to the enumeration's underlying type, is out of range (if the underlying type is fixed, the range is the range of the type. If the underlying type is not fixed, the range is all values possible for the smallest bit field large enough to hold all enumerators of the target enumeration)

From: http://en.cppreference.com/w/cpp/language/static_cast

What happens when you execute undefined behavior?

- Case 1: Program breaks immediately (segfault, math exception)
- Case 2: Program continues, but will fail later (corrupted RAM, etc.)
- Case 3: Program works as expected
  - However, the UB is a "time bomb" − a latent problem waiting to go off when the compiler, compiler version, or compiler flags is changed
- Case 4: You don't know how to trigger the UB but someone else does

Important trends over the last 25 years:

- UB detection tools have been getting better
  - Starting perhaps with Purify in the early 1990s
  - More about these later
- Compilers have been getting cleverer at exploiting UB to improve code generation
  - Making the time bombs go off
  - More about this soon
- Security has become a primary consideration in software development

- But of course legacy C and C++ contain plenty of UB
- What can we do about that?
  - Sane option 1: Go back and fix the old code
    - Expensive...
  - Sane option 2: Stop setting off time bombs by making optimizers more aggressive
  - Not as sane: Keep making optimizers more aggressive while also not investing in maintenance of older codes

```
int foo (int x) {
    return (x + 1) > x;
}
int main() {
    printf("%d\n", (INT_MAX + 1) > INT_MAX);
    printf("%d\n", foo(INT_MAX));
    return 0;
}
```

```
$ gcc -O2 foo.c ; ./a.out
0
1
```

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

```
$ clang -O foo.c ; ./a.out
1 2
```

Without -DDEBUG

```
void foo(char *p) {
#ifdef DEBUG
  printf("%s\n", p);
#endif
  if (p)
    bar(p);
}
```

```
_foo:
    testq   %rdi, %rdi
    je      L1
    jmp     _bar
L1: ret
```

With -DDEBUG

```
void foo(char *p) {
#ifdef DEBUG
  printf("%s\n", p);
#endif
  if (p)
    bar(p);
}
```

```
_foo:
    pushq   %rbx
    movq    %rdi, %rbx
    call    _puts
    movq    %rbx, %rdi
    popq    %rbx
    jmp     _bar
```

```
void foo(int *p,
         int *q,
         size_t n) {
    memcpy(p, q, n);
    if (!q)
      abort();
}
```

```
_foo:
        jmp     memcpy
```

Optimization is valid even when n == 0

There are a lot more UBs!

- Strict aliasing
  - If two pointers refer to different types, compiler may assume they don't refer to the same object
  - Vast majority of programs violate strict aliasing
- Infinite loops that don't contain side-effecting operations are UB
  - Common case: compiler causes the loop to exit
  - In C11 the compiler cannot terminate an infinite loops whose controlling expression is a constant expression
    - So then we can at least rely on **`while (1)`** …

- Effects of UB can precede the first undefined operation
  - Potentially undefined operations are not seen as side-effecting
  - Compiler can move a crashing operation in front of a debug printout!
  - This is explicit in the C++ standard:

5 A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

So what can do we about UB in C and C++?

- Static detection
- Dynamic detection
- Mitigation

Static analysis

- Enable and heed compiler warnings
  - Use **`-Werror`**
- Code reviewers should be thinking about UB
- Run unsound static analysis tools
  - Coverity
  - Clang static analyzer
  - Lots more
- Run sound static analysis tools
  - Polyspace Analyzer
  - Frama-C / TIS Analyzer

Dynamic Analysis using Clang (and GCC)

- Address Sanitizer (ASan)
  - Memory safety errors
- Undefined Behavior Sanitizer (UBSan)
  - Shift errors, signed integer overflow, alignment issues, missing return statements, etc.
- Memory Sanitizer (MSan)
  - Use of uninitialized storage
- Thread Sanitizer (TSan)
  - Data races, deadlocks

- Dynamic analysis is great, since you get a concrete execution trace
- Dynamic analysis is terrible, since you need test cases to drive concrete execution traces
- Where do we get concrete inputs?
  - Test suites
  - Fuzzers
  - …

Missing dynamic analysis tools

- Strict aliasing
- Non-terminating loops
- Unsequenced side effects
  - In a function argument list
  - In an expression

UB Mitigation

- Linux compiles with **`-fno-delete-null-pointer-checks`**
- MySQL compiles with **`-fwrapv`**
- Many programs compile with **`-fno-strict-aliasing`**
- Parts of Android use UBSan in production
- Chrome is built with control flow integrity (CFI) enabled in x86-64
  - Provided by recent LLVMs
  - Overhead < 1%

Issues with UB mitigation

- Solutions aren't standardized or portable
- There's no mitigation for concurrency errors
- Memory safety error mitigation tends to be expensive and may break code
  - And ASan is not a hardening tool
- UBSan can be configured as a hardening tool
  - Software developers need to decide if they want to turn a potential exploit into a crash

Summary

- UB is still a serious problem
  - It's probably too late to fix the C or C++ standard
  - There's hope for safer dialects
- Tools for managing UB are steadily improving
- All of static detection, dynamic detection, and mitigation should be used
  - Software testing remains extremely difficult

Thanks!