

# Undefined Behavior And u



Thomas Peters,  
Feb 26, 2018

# Outline

- What is Undefined Behavior
- Why does it exist
- Examples
- Why aren't compilers more helpful
- What can we do about it

# C++

- Standardized by International Standards Organization (ISO)

## Standards:

- C++98
- C++03 (small change, mostly bugfixes)
- C++11 (substantial improvement to language)
- C++14 (relatively small)
- C++17 (just born!)

# C++ Standards

- Define what C++ is at an abstract level
- Give some, ***but not all***, rules for how programs must behave.
- Talk in terms of the ***C++ Abstract Machine***, and ***implementations***.

# As-if rule

*“The semantic descriptions in this International Standard define a **parameterized nondeterministic abstract machine**. This International Standard **places no requirement on the structure of conforming implementations**. In particular, they need not copy or emulate the structure of the abstract machine. Rather, **conforming implementations are required to emulate (only) the observable behavior of the abstract machine**.*

*This provision is sometimes called the “**as-if**” rule, because an implementation is free to disregard any requirement of this International Standard as long as the result is as if the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.”*

# Observable behavior

- Accesses (reads and writes) to `volatile` objects occur strictly according to the semantics of the expressions in which they occur.
- At program termination, data written to files is exactly as if the program was executed as written.
- Prompting text which is sent to interactive devices will be shown before the program waits for input.
- Some crap about floating point environment. ZZZZ

# Exceptions to the as-if rule 1:

- **Copy elision.** The compiler may remove calls to move- and copy-constructors and the matching calls to the destructors of temporary objects even if those calls have observable side effects.
- **New Expressions.** the compiler may remove calls to the replaceable allocation functions even if a user-defined replacement is provided and has observable side-effects.

# Exceptions to the as-if rule 2:

*“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).”*

*“Certain other operations are described in this International Standard as undefined (for example, the effect of attempting to modify a const object). [ Note: This International Standard imposes no requirements on the behavior of programs that contain undefined behavior. —end note ]”*





Question:

What is `INT_MAX + 1`?

It is not guaranteed to be `INT_MIN`.

*The result is undefined.*

# Undefined Behavior (UB): The bad

- “Renders the entire program meaningless”
- “Anything can happen”
  - Extreme:
    - Program formats your hard drive
    - Your cat gets pregnant (even if you have no cat)
  - More realistic
    - Program crashes (good case!)
    - Program does what you expected
    - Program gives wrong answers
- *Compilers often don't warn about UB*

*Think of invoking UB as saying **there is an error in your program.***

# Undefined Behavior: The good

*Compilers use UB to generate faster and smaller code, often when optimizations are enabled.*

Instead of saying “this is an error,” compilers will often say “this cannot happen”.


Example:

*// signed integer overflow is UB*

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

Compiler:

Hm, if  $x < \text{INT\_MAX}$ ,  
This always returns  
true, otherwise UB. UB  
allows me to do  
whatever I want, and  
you asked for small fast  
code, so I'll return true  
there too.



*// optimized code is equivalent  
to*

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

*// can avoid with -fwrapv,  
// -ftrapv on gcc, clang*

# (Some!) Examples of undefined behavior

- Memory operations:
  - Null pointer dereference
  - Array access out of bounds
  - Modification of a const object
  - Use of uninitialized variable
  - Use of object after lifetime end
  - Strict aliasing violations
- Integral arithmetic
  - Signed integer overflow
  - Shifting beyond width (eg, `int8_t x; x << 8;`)
  - Integer division by zero
- Converting numeric value to type without sufficient bits to represent it.
- Static initialization with dependencies on other static objects.
- Data races
- Reaching end of value-returning function (other than main) without returning

# Example: strict aliasing

```
float funky_float_abs (float a)
{
    unsigned int temp = *(unsigned int *)&a; // strict aliasing violation
    temp &= 0x7fffffff;
    return *(float *)&temp; // strict aliasing violation
}
```

```
float funky_float_abs (float a)
{
    float temp_float = a;
    unsigned char * temp = (unsigned char *)&temp_float; // OK: char* s may alias anything
    temp[3] &= 0x7f;
    return temp_float;
}
```

```
float funky_float_abs (float a)
{
    unsigned int i;
    float result;
    memcpy (&i, &a, sizeof (unsigned int)); // OK:
    i &= 0x7fffffff;
    memcpy (&result, &i, sizeof (unsigned int));
    return result;
}
```

NB: clang, gcc have  
**-fno-strict-aliasing** flags

# Example: Access out of bounds

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

Compiler: return true in one of  
first four iterations or we hit  
UB.

I'll always return true.



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

# Null pointer access, security vulnerability

```
static void __devexit agnx_pci_remove (struct pci_dev *pdev)
{
    struct ieee80211_hw *dev = pci_get_drvdata(pdev);
    struct agnx_priv *priv = dev->priv;

    if (!dev) return;
    ... do stuff using dev ...
}
```

From a bug in the Linux kernel!

Compiler: If dev is not null, we do stuff with it.  
If dev is null, we dereference it (UB).  
*I can assume dev is never null.*

```
static void __devexit agnx_pci_remove (struct pci_dev *pdev)
{
    struct ieee80211_hw *dev = pci_get_drvdata(pdev);
    struct agnx_priv *priv = dev->priv;
    ... do stuff using dev ...
}
```

Linux kernel now uses  
-fno-delete-null-pointer-checks

# Effect on debugging

```
printf("hello\n");  
printf("world\n");
```

*// must output*

```
hello  
world
```

```
int a, b;
```

```
...
```

```
a++;
```

```
b++;
```

*// compiler is free to reorder (or remove!) operations (unless values are volatile-qualified).*

*// debugging some crashing code*

```
std::cout << "got to here" << std::endl;
```

```
suspicious_function_call();
```

Say program crashes without printing “got to here”

What can we conclude?

*We can't conclude that the cause of the crash is before the function.*



# Interacting compiler optimizations


<pre>void contains_null_check(int *P) {     int dead = *P;     if (P == 0)         return;     *P = 4; }</pre>	<pre>void contains_null_check(int *P) {     int dead = *P;     if (P == 0)         return;     *P = 4; }</pre>
Dead code elimination	Redundant null check elimination
<pre>void contains_null_check_after_dce(int *P) {     if (P == 0)         return;     *P = 4; }</pre>	<pre>void contains_null_check_after_rnce(int *P) {     int dead = *P;     if (false)         return;     *P = 4; }</pre>
Redundant null check elimination	Dead code elimination
<pre>void contains_null_check_after_dce_after_rnce(int *P) {     if (P == 0) <i>// not redundant</i>         return;     *P = 4; }</pre>	<pre>void contains_null_check_after_rnce_after_dce(int *P) {     *P = 4; }</pre>

# UB can actually format your hard drive

```
#include <cstdlib>

static void (*FP)() = 0;
static void impl() {
    system("rm -rf /");
}
void set() {
    FP = impl;
}
void call() {
    FP();
}
```

Compiler: in call, either FP is NULL (UB) or it has been set to impl. I can assume it's always set to impl.



```
#include <cstdlib>

static void impl() {
    system("rm -rf /");
}
static void (*FP)() = impl;
void set() {
    FP = impl;
}
void call() {
    FP();
}
```

# What can we do about UB?

- Educate yourself. Read CppCoreGuidelines, high-quality C++ books.
- Be paranoid, and look out for UB during code reviews
- Crank up your warnings, and listen to them: Wall, Wextra, Wpedantic
- Static analysis:
  - cppCheck (not a whole lot more than warnings already give you)
  - Clang-tidy (warnings, some static analysis)
  - Clang-check (static analysis beyond warnings)
  - Visual studio??
  - Commercial tools (CodeSonar \$\$\$)
- Dynamic (runtime) analysis
  - Valgrind
  - Sanitizers: ubsan, address sanitizer, memory sanitizer (clang, some gcc support)
- If you're a praying person, do that.
- *Consider another programming language*
  - Java, Python are fine for many tasks, and have more predictable behavior on errors.
  - Rust: a young, new systems language designed specifically with safety in mind.

# Why don't compilers do a better job warning UB?

1. Just like halting problem, detecting all UB is impossible.
2. Warning on all potential UB instances would produce too many warnings to be useful, most of which are false positives.
3. People don't want warnings for dead code.
4. It's extremely hard to show to a user how a sequence of optimizations led to potential UB.
5. Compile times. (Running static analysis takes a long time)

# Less evil siblings of UB

1. **Implementation-defined.** The behavior of the program varies between implementations, and the conforming implementation must document the effects of each behavior. These behaviors constitute the parameters of the abstract machine. Examples:
  - a. Size of `size_t`
  - b. bitness of `char` (!)
  - c. Signedness of `char`.
  - d. `what()` strings in `std` exceptions.
2. **Unspecified.** the behavior of the program varies between implementations and the conforming implementation is not required to document the effects of each behavior. Where possible the standard describes a set of allowable behaviors. These behaviors constitute the nondeterministic aspects of the abstract machine. Examples:
  - a. Order of evaluation for function arguments.

# Summary

- UB are behaviors outside of the rules of C++.
- Don't do what Donny Dont does: using undefined behavior in a program is almost always an error.
- Compilers use undefined behavior to produce smaller or faster code.

Stay safe out there.

Questions?

# References

- <https://www.youtube.com/watch?v=KogY50HSuQg> (Patrice Roy, “Which Machine am I coding to”, cppcon 2017)
- <http://en.cppreference.com/w/cpp/language/ub>
- <http://port70.net/~nsz/c/c99/n1256.html#J.2> (list of UB in C)
- <http://blog.lvm.org/2011/05/what-every-c-programmer-should-know.html> “What Every C Programmer Should Know About Undefined Behavior”
- <https://blog.regehr.org/archives/213> “A Guide to Undefined Behavior in C and C++”