Undefined behavior

In computer programming, **undefined behavior** (**UB**) is the result of executing a program whose behavior is prescribed to be unpredictable, in the language specification of the programming language in which the source code is written. This is different from unspecified behavior, for which the language specification does not prescribe a result, and implementation-defined behavior that defers to the documentation of another component of the platform (such as the ABI or the translator documentation).

In the C programming community, undefined behavior may be humorously referred to as "**nasal demons**", after a comp.std.c post that explained undefined behavior as allowing the compiler to do anything it chooses, even "to make demons fly out of your nose".[1]

Overview

Some programming languages allow a program to operate differently or even have a different control flow from the source code, as long as it exhibits the same user-visible side effects, *if undefined behavior never happens during program execution*. Undefined behavior is the name of a list of conditions that the program must not meet.

In the early versions of C, undefined behavior's primary advantage was the production of performant compilers for a wide variety of machines: a specific construct could be mapped to a machine-specific feature, and the compiler did not have to generate additional code for the runtime to adapt the side effects to match semantics imposed by the language. The program source code was written with prior knowledge of the specific compiler and of the platforms that it would support.

However, progressive standardization of the platforms has made this less of an advantage, especially in newer versions of C. Now, the cases for undefined behavior typically represent unambiguous bugs in the code, for example indexing an array outside of its bounds. By definition, the runtime can assume that undefined behavior never happens; therefore, some invalid conditions do not need to be checked against. For a compiler, this also means that various program transformations become valid, or their proofs of correctness are simplified; this allows for various kinds of optimizations whose correctness depend on the assumption that the program state never meets any such condition. The compiler can also remove explicit checks that may have been in the source code, without notifying the programmer; for example, detecting undefined behavior by testing whether it happened is not guaranteed to work, by definition. This makes it hard or impossible to program a portable fail-safe option (non-portable solutions are possible for some constructs).

Current compiler development usually evaluates and compares compiler performance with benchmarks designed around micro-optimizations, even on platforms that are mostly used on the general-purpose desktop and laptop market (such as amd64). Therefore, undefined behavior provides ample room for compiler performance improvement, as the source code for a specific source code statement is allowed to be mapped to anything at runtime.

For C and C++, the compiler is allowed to give a compile-time diagnostic in these cases, but is not required to: the implementation will be considered correct whatever it does in such cases, analogous to don't-care terms in digital logic. It is the responsibility of the programmer to write code that never invokes undefined behavior, although compiler implementations are allowed to issue diagnostics when this happens. Compilers nowadays have flags that enable such diagnostics, for example, `-fsanitize=undefined` enables the "undefined behavior sanitizer" (UBSan) in gcc 4.9[2] and in clang. However, this flag is not the default and enabling it is a choice of the person who builds the code.

Under some circumstances there can be specific restrictions on undefined behavior. For example, the instruction set specifications of a CPU might leave the behavior of some forms of an instruction undefined, but if the CPU supports memory protection then the specification will probably include a blanket rule stating that no user-accessible instruction may cause a hole in the operating system's security; so an actual CPU would be permitted to corrupt user registers in response to such an instruction, but would not be allowed to, for example, switch into supervisor mode.

The runtime platform can also provide some restrictions or guarantees on undefined behavior, if the toolchain or the runtime explicitly document that specific constructs found in the source code are mapped to specific well-defined mechanisms available at runtime. For example, an interpreter may document a particular behavior for some operations that are undefined in the language specification, while other interpreters or compilers for the same language may not. A compiler produces executable code for a specific ABI, filling the semantic gap in ways that depend on the compiler version: the documentation for that compiler version and the ABI specification can provide restrictions on undefined behavior. Relying on these implementation details makes the software non-portable, but portability may not be a concern if the software is not supposed to be used outside of a specific runtime.

Undefined behavior can result in a program crash or even in failures that are harder to detect and make the program look like it is working normally, such as silent loss of data and production of incorrect results.

Benefits

Documenting an operation as undefined behavior allows compilers to assume that this operation will never happen in a conforming program. This gives the compiler more information about the code and this information can lead to more optimization opportunities.

An example for the C language:

```c
int foo(unsigned char x)
{
     int value = 2147483600;  /* assuming 32-bit int and 8-bit char */
     value += x;
     if (value < 2147483600)
        bar();
     return value;
}
```

The value of `x` cannot be negative and, given that signed integer overflow is undefined behavior in C, the compiler can assume that `value < 2147483600` will always be false. Thus the `if` statement, including the call to the function `bar`, can be ignored by the compiler since the test expression in the `if` has no side effects and its condition will never be satisfied. The code is therefore semantically equivalent to:

```
int foo(unsigned char x)
{
     int value = 2147483600;
     value += x;
     return value;
}
```

Had the compiler been forced to assume that signed integer overflow has *wraparound* behavior, then the transformation above would not have been legal.

Such optimizations become hard to spot by humans when the code is more complex and other optimizations, like inlining, take place. For example, another function may call the above function:

```
void run_tasks(unsigned char *ptrx) {
    int z;
    z = foo(*ptrx);
    while (*ptrx > 60) {
        run_one_task(ptrx, z);
    }
}
```

The compiler is free to optimize away the `while`-loop here by applying value range analysis: by inspecting `foo()`, it knows that the initial value pointed to by `ptrx` cannot possibly exceed 47 (as any more would trigger undefined behavior in `foo()`); therefore, the initial check of `*ptrx > 60` will always be false in a conforming program. Going further, since the result `z` is now never used and `foo()` has no side effects, the compiler can optimize `run_tasks()` to be an empty function that returns immediately. The disappearance of the `while`-loop may be especially surprising if `foo()` is defined in a separately compiled object file.

Another benefit from allowing signed integer overflow to be undefined is that it makes it possible to store and manipulate a variable's value in a processor register that is larger than the size of the variable in the source code. For example, if the type of a variable as specified in the source code is narrower than the native register width (such as `int` on a 64-bit machine, a common scenario), then the compiler can safely use a signed 64-bit integer for the variable in the machine code it produces, without changing the defined behavior of the code. If a program depended on the behavior of a 32-bit integer overflow, then a compiler would have to insert additional logic when compiling for a 64-bit machine, because the overflow behavior of most machine instructions depends on the register width.[3]

Undefined behavior also allows more compile-time checks by both compilers and static program analysis.

Risks

C and C++ standards have several forms of undefined behavior throughout, which offer increased liberty in compiler implementations and compile-time checks at the expense of undefined run-time behavior if present. In particular, the ISO standard for C has an appendix listing common sources of undefined behavior.[4] Moreover, compilers are not required to diagnose code that relies on undefined behavior. Hence, it is common for programmers, even experienced ones, to rely on undefined behavior either by mistake, or simply because they are not well-versed in the rules of the language that can span hundreds of pages. This can result in bugs that are exposed when a different compiler, or different settings, are used. Testing or fuzzing with dynamic undefined behavior checks enabled, e.g., the Clang sanitizers, can help to catch undefined behavior not diagnosed by the compiler or static analyzers.[5]

Undefined behavior can lead to security vulnerabilities in software. For example, buffer overflows and other security vulnerabilities in the major web browsers are due to undefined behavior. When GCC's developers changed their compiler in 2008 such that it omitted certain overflow checks that relied on undefined behavior, CERT issued a warning against the newer versions of the compiler.[6] Linux Weekly News pointed out that the same behavior was observed in PathScale C, Microsoft Visual C++ 2005 and several other compilers;[7] the warning was later amended to warn about various compilers.[8]

Examples in C and C++

The major forms of undefined behavior in C can be broadly classified as:[9] spatial memory safety violations, temporal memory safety violations, integer overflow, strict aliasing violations, alignment violations, unsequenced modifications, data races, and loops that neither perform I/O nor terminate.

In C the use of any automatic variable before it has been initialized yields undefined behavior, as does integer division by zero, signed integer overflow, indexing an array outside of its defined bounds (see buffer overflow), or null pointer dereferencing. In general, any instance of undefined behavior leaves the abstract execution machine in an unknown state, and causes the behavior of the entire program to be undefined.

Attempting to modify a string literal causes undefined behavior:[10]

```
char *p = "wikipedia";  // valid C, deprecated in C++98/C++03, ill-formed as of C++11
p[0] = 'W';  // undefined behavior
```

Integer division by zero results in undefined behavior:[11]

```
int x = 1;
return x / 0;  // undefined behavior
```

Certain pointer operations may result in undefined behavior:[12]

```
int arr[4] = {0, 1, 2, 3};
int *p = arr + 5;  // undefined behavior for indexing out of bounds
p = NULL;
int a = *p;        // undefined behavior for dereferencing a null pointer
```

In C and C++, the relational comparison of pointers to objects (for less-than or greater-than comparison) is only strictly defined if the pointers point to members of the same object, or elements of the same array.[13] Example:

```
int main(void)
{
  int a = 0;
  int b = 0;
  return &a < &b;  /* undefined behavior */
}
```

Reaching the end of a value-returning function (other than `main()`) without a return statement results in undefined behavior if the value of the function call is used by the caller:[14]

```
int f()
{
}  /* undefined behavior if the value of the function call is used*/
```

Modifying an object between two sequence points more than once produces undefined behavior.[15] There are considerable changes in what causes undefined behavior in relation to sequence points as of C++11.[16] Modern compilers can emit warnings when they encounter multiple unsequenced modifications to the same object.[17][18] The following example will cause undefined behavior in both C and C++.

```
int f(int i) {
  return i++ + i++;  /* undefined behavior: two unsequenced modifications to i */
}
```

When modifying an object between two sequence points, reading the value of the object for any other purpose than determining the value to be stored is also undefined behavior.[19]

```
a[i] = i++;  // undefined behavior
printf("%d %d\n", ++n, power(2, n));  // also undefined behavior
```

In C/C++ bitwise shifting a value by a number of bits which is either a negative number or is greater than or equal to the total number of bits in this value results in undefined behavior. The safest way (regardless of compiler vendor) is to always keep the number of bits to shift (the right operand of the `<<` and `>>` bitwise operators) within the range: [`0, sizeof value * CHAR_BIT - 1`] (where `value` is the left operand).

```
int num = -1;
unsigned int val = 1 << num;  // shifting by a negative number - undefined behavior

num = 32;  // or whatever number greater than 31
val = 1 << num;  // the literal '1' is typed as a 32-bit integer - in this case shifting by more than
31 bits is undefined behavior

num = 64;  // or whatever number greater than 63
unsigned long long val2 = 1ULL << num;  // the literal '1ULL' is typed as a 64-bit integer - in this
case shifting by more than 63 bits is undefined behavior
```

Examples in Rust

While undefined behavior is never present in safe Rust, it is possible to invoke undefined behavior in unsafe Rust in many ways.[20] For example, creating an invalid reference (a reference which does not refer to a valid value) invokes immediate undefined behavior:

```
fn main() {
    // The following line invokes immediate undefined behaviour.
    let _null_reference: &i32 = unsafe { std::mem::zeroed() };
}
```

It is not necessary to use the reference; undefined behavior is invoked merely from the creation of such a reference.

See also

- Compiler
- Halt and Catch Fire
- Unspecified behavior

References

1. "nasal demons" (http://catb.org/jargon/html/N/nasal-demons.html). *Jargon File*. Retrieved 12 June 2014.
2. *GCC Undefined Behavior Sanitizer – ubsan* (https://developers.redhat.com/blog/2014/10/16/gcc-undefined-behavior-sanitizer-ubsan/)
3. "A bit of background on compilers exploiting signed overflow" (https://gist.github.com/rygorous/e0f055bfb74e3d5f0af20690759de5a7#file-gistfile1-txt-L166).
4. ISO/IEC 9899:2011 §J.2.
5. John Regehr (19 October 2017). "Undefined behavior in 2017, cppcon 2017" (https://www.youtube.com/watch?v=v1COuU2vU_w). *YouTube*.
6. "Vulnerability Note VU#162289 — gcc silently discards some wraparound checks" (https://web.archive.org/web/20080409224149/http://www.kb.cert.org/vuls/id/162289). *Vulnerability Notes Database*. CERT. 4 April 2008. Archived from the original (http://www.kb.cert.org/vuls/id/162289) on 9 April 2008.
7. Jonathan Corbet (16 April 2008). "GCC and pointer overflows" (http://lwn.net/Articles/278137/). *Linux Weekly News*.
8. "Vulnerability Note VU#162289 — C compilers may silently discard some wraparound checks" (http://www.kb.cert.org/vuls/id/162289). *Vulnerability Notes Database*. CERT. 8 October 2008 [4 April 2008].
9. Pascal Cuoq and John Regehr (4 July 2017). "Undefined Behavior in 2017, Embedded in Academia Blog" (https://blog.regehr.org/archives/1520).
10. ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages – C++ §2.13.4 String literals [lex.string]* para. 2
11. ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages – C++ §5.6 Multiplicative operators [expr.mul]* para. 4
12. ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages - C++ §5.7 Additive operators [expr.add]* para. 5
13. ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages – C++ §5.9 Relational operators [expr.rel]* para. 2
14. ISO/IEC (2007). *ISO/IEC 9899:2007(E): Programming Languages – C §6.9 External definitions* para. 1
15. ANSI X3.159-1989 *Programming Language C*, footnote 26
16. "Order of evaluation - cppreference.com" (http://en.cppreference.com/w/cpp/language/eval_order). *en.cppreference.com*. Retrieved 9 August 2016.

17. "Warning Options (Using the GNU Compiler Collection (GCC))" (https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html). *GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)*. Retrieved 2021-07-09.
18. "Diagnostic flags in Clang" (https://clang.llvm.org/docs/DiagnosticsReference.html#wunsequenced). *Clang 13 documentation*. Retrieved 2021-07-09.
19. ISO/IEC (1999). *ISO/IEC 9899:1999(E): Programming Languages – C §6.5 Expressions* para. 2
20. "Behavior considered undefined" (https://doc.rust-lang.org/reference/behavior-considered-undefined.html). *The Rust Reference*. Retrieved 2022-11-28.

Further reading

- Peter van der Linden, *Expert C Programming*. ISBN 0-13-177429-8
- UB Canaries (https://blog.regehr.org/archives/1234) (April 2015), John Regehr (University of Utah, USA)
- Undefined Behavior in 2017 (https://blog.regehr.org/archives/1520) (July 2017) Pascal Cuoq (TrustInSoft, France) and John Regehr (University of Utah, USA)

External links

- Corrected version of the C99 standard (https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf). See at section 6.10.6 for #pragma