Is undefined behaviour preserved?

ISO/IEC JTC1 SC22 WG21 - D1093R1

Working Group: Evolution, Library Evolution, Undefined Behaviour

Date: 2018-06-03

Andrew Bennieston <a.j.bennieston@gmail.com>

Jonathan Coe <jonathanbcoe@gmail.com>

Daven Gahir <daven.gahir@gmail.com>

Thomas Russell <thomas.russell97@gmail.com>

> *UB or not UB? That is the question: whether 'tis nobler in the mind to suffer the slings and arrows of undefined behaviour, or to take arms against a sea of troubles and by defining: end them.*
>
> – William Shakespeare, Hamlet (adapted).

TL;DR

Undefined behaviour can be exploited to optimise code and used by compiler and library implementations to help find bugs. Can we rely on such optimisations and bug-squashing abilities after a compiler upgrade?

Introduction

Should C++ guarantee that undefined behaviour remains undefined behaviour as the language and library evolve?

We have recently seen papers that propose rendering currently undefined behaviour as well-defined [1] [2]. In the ensuing discussions, concerns were raised about the possibility of degraded run-time performance (e.g. due to missed optimisation opportunities) and lost ability to detect bugs (e.g. due to tools like `ubsan` being increasingly constrained). Rather than have precedent determined by a small number of concrete cases, we would like to discuss more generally the issue of changes to the language and library that aim to eliminate undefined behaviour.

In this paper, following the spirit of [3] and [4], we invite the combined evolution groups to discuss, if not determine, (non-binding) policy on preserving or eliminating undefined behaviour.

Contracts, preconditions and postconditions

Contract-based-programming is a software design method where formal requirements and guarantees are given for functions. Contract design for C++ is described in [5] and [6] and its impact considered in [7].

From the proposed wording in [6]:

"A precondition is a predicate that should hold upon entry into a function. It expresses a function's expectation on its arguments and/or the state of objects that may be used by the function."

"A postcondition is a predicate that should hold upon exit from a function. It expresses the conditions that a function should ensure for the return value and/or the state of objects that may be used by the function."

A function with preconditions is said to have a *narrow contract*. Violating the preconditions on such a function may result in an ill-formed program and associated compile-time-diagnostics or in undefined behaviour.

When there are no preconditions on a function's arguments, the function is said to have a *wide contract*. There may be input values for wide-contract-functions that result in exceptions being thrown but such behaviour is always well-defined. `std::vector` has `operator[]` and `at` to perform index-access with narrow and wide contracts respectively.

Changes to function contracts

In an updated version of the C++ Standard we may wish to consider making changes to a function's preconditions and postconditions.

Making preconditions narrower would be a silent breaking change: a previously valid program would now invoke undefined behaviour. We would expect such a proposed change to be rejected.

Widening postconditions would similarly be a silent breaking change: a previously valid program that relied on the postconditions of one function to satisfy the preconditions of another would now invoke undefined behaviour. We would similarly expect such a proposed change to be rejected.

Widening preconditions would not render any existing program invalid or undefined. We would expect such a change to be accepted so long as it was not accompanied by widening of postconditions.

Making postconditions narrower would similarly not render any existing program invalid or undefined. We would expect such a change to be accepted.

Widening preconditions and narrowing postconditions may render some existing programs less-than optimally efficient as they may contain run-time checks for behaviour that is now guaranteed.

There may be other factors to consider though. People may be *relying* on undefined behaviour for trapping errors or for optimisation. Additionally, if we were to narrow a postcondition, by now making a function throw instead of, say, return null, this would potentially have significant impact on existing programs, but not make them invalid or undefined. In this paper, we do not address this scenario, as we wish to focus specifically on undefined behaviour.

Sanitizers and assertions

The undefined behaviour sanitizers from GCC [8] and Clang [9] can be used to produce an instrumented build in which some instances of undefined behaviour will be detected and the program terminated with a helpful message.

Standard library implementations can be augmented with debug checks and assertions to ensure that preconditions are true. For instance, calling `std::vector::operator[](size_t i)` with `i` greater than the size of the vector will be caught in a debug build using Microsoft's Standard Library implementation.

Builds with sanitizers and active assertions are commonly used by engineers to identify and eliminate bugs.

Case studies

The following case studies are recent examples of proposed changes to undefined behaviour. Both cases would be considered acceptable by the contract-based criteria we outlined above but have led to considerable discussion.

Widening a precondition for `std::string_view`

P0903 [1] proposes to widen the contract of the `string_view(const char*)` constructor to make `string_view((const char*)nullptr)` well-defined. In this paper, we take no position on whether this should be adopted, but present some of the arguments raised for and against adoption from the perspective of widening the interface.

Widening the contract of `string_view`'s pointer constructor is a non-breaking change as it does not result in the widening of postconditions: `string_view` can already be constructed in a state where it has a `NULL` data member using the pointer and size constructor (`string_view(const char*, size_t)`).

Making previously undefined behaviour well-defined will make some bugs harder to find as it will no longer be possible to assert that the pointer in non-null.

The last point is sufficiently subtle for an example to be illuminating. Consider the following code:

```
std::string processName(std::string_view username);

std::string getUserName(const UserConfig& user_config) {
  const char* username = nullptr;

  switch (user_config.user_type) {
    case UserType::FOO:
      // External library C function
      getFooName(user_config.uuid, &username);
      break;
    case UserType::BAR:
      // Programmer error: forgot to call getBarName
      break;
  }

  return processName(username);
}
```

Let us assume that `getFooName` and `getBarName` always set their second parameter to point to a valid C string (possibly empty, i.e. `"\0"`).

In this case, the programmer forgot to handle the `BAR` case properly, and the call to `processName` may construct a `string_view` with a `nullptr` argument. In the current specification, passing `nullptr` to the single-parameter constructor for `string_view` violates a precondition and may result in undefined behaviour when attempting to calculate the length of the string view (which will involve dereferencing the pointer). This provides an opportunity for a library implementation to emit diagnostics that could guide the developer towards the source of the problem.

By making this constructor valid, widening the contract to internally reinterpret it as a call to `string_view(nullptr, 0)`, we would eliminate the possibility for the library to flag this for attention, and instead produce an empty string view, a situation that is much harder to debug as the cause and effect may be separated by some considerable distance or time.

The alternative argument to the above, is that it is wrong to use postconditions to detect this kind of logic / programming error. Instead a small unit test could not only cover the error case above, but also other errors that would be undetectable by a static analysis tool. This would eliminate the argument of error detection, and leave the path open for changes to the specification of `string_view`.

Defining the behaviour for signed integer overflow

P0907R0 [2] originally proposed to make signed integer overflow well-defined such that it behaves as for unsigned integers on overflowing operations (i.e. over-

flow in the positive direction wraps around from the maximum integer value for the type back to the minimum and vice versa for overflow in the opposite direction). This was subsequently removed from the proposal following various concerns raised from EWG, SG6 and SG12. Below we present a quick overview of the reasons for removal of the sub-proposal defining signed integer overflow.

According to the earlier contract-based arguments, making integer overflow well-defined is a non-breaking change, as it widens preconditions and does not further narrow postconditions - there is no previously valid program that would be rendered invalid by making integer overflow well defined.

The primary complaint against defining overflow for signed integers was lost optimisation opportunities and the subsequent expected performance degradation. Modern compilers take advantage of the currently undefined behaviour on signed integer overflow for a variety of optimisations.

Possibly the most crucial of the currently permitted optimisations is loop analysis. A simple `for` loop such as the one below would be adversely affected:

```
signed int foo(signed int i) noexcept
{
  signed int j, k = 0;
  for (j = i; j < i + 10; ++j) ++k;
  return k;
}
```

We might expect that the function `foo` could be trivially reduced to a simple `return 10` statement during a flow-analysis optimisation pass. Indeed, with the current language rules, this is what most modern compilers will emit. However, under the changes proposed in P0907R0 [2] this would no longer be a valid optimisation as there are some inputs which would overflow.

There are many other optimisation opportunities that are similarly reliant on the undefined behaviour of signed integer overflow. Below is an (incomplete) summary of other optimisations gathered from [10]:

- `(x * c) == 0` can be optimised to `x == 0`, eliding the multiplication.
- `(x * c_1) / c_2` can be optimised to `x * (c_1 / c_2)` if `c_1` is divisible by `c_2`.
- `(-x) / (-y)` can be optimised to `x / y`.
- `(x + c) < x` can be optimised to `false` if `c > 0` or `true` otherwise.
- `(x + c) <= x` can be optimised to `false` if `c >= 0` or `true` otherwise.
- `(x + c) > x` can be optimised to `true` if `c >= 0` and `false` otherwise.
- `(x + c) >= x` can be optimised to `true` if `c > 0` and `false` otherwise.
- `-x == -y` can be optimised to `x == y`.
- `x + c > y` can be optimised to `x + (c - 1) >= y`.
- `x + c <= y` can be optimised to `x + (c - 1) < y`.
- `(x + c_1) == c_2` can be optimised to `x == (c_2 - c_1)`.
- `(x + c_1) == (y + c_2)` can be optimised to `x == (y + (c_2 - c_1))` if `c_1 <= c_2`.

- Various value-range specific optimisations such as:
  - Changing comparisons `x < y` to `true` or `false`.
  - Changing `min(x,y)` or `max(x,y)` to `x` or `y` if the ranges do not overlap.
  - Changing `abs(x)` to `x` or `-x` if the range does not cross 0.
  - Changing `x / c` to `x >> log2(c)` if `x > 0`
  - Changing `x % c` to `x & (c-1)` if `x > 0` and the constant `c` is a power of 2.

Polls

- Undefined behaviour should be preserved from one version of the C++ Standard to the next.
- Compiler / library diagnostics that allow undefined behaviour to be trapped should be considered when determining if a new feature is non-breaking.
- Compiler / library optimisations that exploit undefined behaviour should be considered when determining if a new feature is non-breaking.

Acknowledgements

We would like to thank Walter Brown for extensive feedback on an early draft of this paper, and the members of the BSI, discussion with whom led to this paper being drafted.

References

- [1] P0903R1 – Define `basic_string_view(nullptr)`, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0903r1.pdf
- [2] P0907R0 – Signed Integers are Two's Complement, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0907r0.html
- [3] P0684R2 – C++ Stability, Velocity and Deployment Plans, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0684r2.pdf
- [4] P0921R0, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0921r0.pdf
- [5] P0380R1 – A Contract Design, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0380r1.
- [6] P0542R1 – Support for contract based programming in C++, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0542r1.html
- [7] P0788R0 – Standard Library Specification in a Concepts and Contracts World, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0788r0.pdf
- [8] GCC Instrumentation Options, https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
- [9] Clang Undefined Behaviour Sanitizer, https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

- [10] How undefined signed overflow enables optimizations in GCC, https://kristerw.blogspot.co.uk/2016/02/how-undefined-signed-overflow-enables.html