Document Number: P2234R1
Date: 2021-02-13
Reply-to: Scott Schurr: s dot scott dot schurr at gmail dot com
Author: Scott Schurr
Audience: SG12 and EWGI

Consider a UB and IF-NDR Audit

Abstract: In the C++ language undefined behavior (UB) and ill-formed no diagnostic required (IF-NDR) situations set traps for intermediate programmers. These traps tend to be subtle, poorly documented, and hard to debug.

Of course it is not possible, or even desirable, to remove all UB and IF-NDR situations. But there may be ways that the committee can reduce its occurrence in the standard without compromising performance. This paper:

- Explores motivations for these kinds of changes, and
- Suggests a potential process for identifying and introducing those changes.

It also includes a speculative appendix which describes the kinds of changes that might be considered and potentially adopted as a result of the audit.

Quotes from and references to the **Standard for Programming Language C++** all come from the C++20 Draft International Standard ISO/IEC DIS 14882:2020[1].

Revision History

Revision 0

The initial revision uses P1407R1, "Tell Programmers About Signed Integer Overflow Behavior"[2], as a starting point. But the new document has significantly larger scope than P1470R1 and changes its approach to possible solutions.

Revision 1

Revision 0 led to confusion about what the paper was asking for. Revision 1 restructured the paper so the more speculative part was moved to an appendix. The speculative part was also refined based on comments by reviewers.

Why Look at All of IF-NDR and UB?

Taken in isolation, each instance of UB or IF-NDR is a small thing. Additionally, individuals often have firmly held opinions regarding UB and IF-NDR, both in general and regarding specific cases.

Individual papers addressing single cases of UB and IF-NDR would probably be considered "small" papers, which are discouraged by P0559R0, Operating Principles for Evolving C++ [3]. Furthermore P2000R1, Direction for ISO C++ [4] also discourages '… isolated "cute" proposals.'

But, in the author's opinion, the total mass of all UB and IF-NDR is a significant problem for intermediate users of the language. If small isolated proposals are not acceptable then perhaps a larger proposal that suggests auditing all of UB and IF-NDR can be a successful approach.

A Motivating Example

Imagine you are a programmer with a background in electrical engineering. You have never worked on compiler internals or C++ standard libraries. You are developing an embedded product on a small team for a small company. You are coding a helper function that clips on integer overflow. Which of these two implementations do you choose?

```
int
add_100_without_wrap (int a)
{
  using namespace std;
  int const ret = a + 100;
  if (ret < a)
    return
      numeric_limits<int>::max();
  return ret;
}
```

```
unsigned int
add_100_without_wrap (unsigned int a)
{
  using namespace std;
  unsigned int const ret = a + 100u;
  if (ret < a)
    return
      numeric_limits<unsigned int>::max();
  return ret;
}
```

You, as a member of the C++ Standards Committee, of course start out by saying, "I would never write that code." Yes, but you were asked to have some imagination. If you had to pick the code that would behave in the expected fashion, you'd pick the code on the right. That's because you know that the code on the left contains undefined behavior.

Pause and think about how subtle the distinction is between the two code snippets. The choice between using a signed or unsigned integer has silently signaled the optimizer whether or not the programmer's code may be removed.

Additionally, debugging the error which *may* be introduced at high optimization could be extremely difficult.

- Any logging statements added inside the conditional may be removed by the optimizer.
- When the optimization level is reduced so the code can run easily in the debugger then the problem may go away.

A beginner, someone unacquainted with how two's complement hardware works, would never have written this code.

An expert in C++, someone who knows about the many aspects of undefined behavior in C and C++, would never have written this code. And, if an intermediate programmer asked a C++ expert what was going on, the expert would identify the problem in short order.

The unassisted intermediate programmer, on the other hand, is in for a long road of debugging, cursing, and confusion.

How About Tools Like ASan and UBSan?

You betcha. These runtime tools are really helpful when they can be applied. However there are plenty of situations where C++ is used and these tools cannot be used effectively.

- Many embedded platforms do not have these tools available.
- All these tools have runtime and memory overhead. So they may be hard or impossible to use in memory or time constrained environments.
- The undefined behavior must be exercised while the analyzer is running in order for the analyzer to detect the problem. If the problem is hard to exercise then it will stay hidden.

So the biggest barrier for many intermediate programmers is that these tools are either not available on their platforms or will not run in their environments.

Still, there are static analyzers which impose no runtime overhead. And turning on all warnings in the compiler helps, since this acts as a form of limited static analysis.

But, even where these tools are available and usable, the second hurtle is that programmers must be aware of the availability and usefulness of these tools before they will use them. In effect, by relying on these tools to detect undefined behavior we're already biasing our efforts toward helping expert C++ programmers. Experts are the ones who know about and understand the value of these tools. The experts are not the people that need the help. It's the intermediate programmers who have problems that they are not even aware of. There are a surprising number of programming shops that don't even turn on all compiler warnings. It's not reasonable to expect such shops to sink a lot of additional effort into sanitizers when they don't even use all the tools already easily at their fingertips.

So in summary, yes, tools outside of the standard can help tame undefined behavior. Yes, since undefined behavior will never be eliminated from the standard, these tools need to continue to improve and gain wider adoption. However when one bit of UB or IF-NDR is removed from the standard, then that particular pitfall has been removed from all consideration. Rather than belatedly patching around the problem with the help of a tool, we can entirely remove that one problem. Removing the problem, when possible, is the stronger position.

Undefined Behavior is Ubiquitous

As shown earlier, the undefined behavior of signed integer overflow can be confusing. Does undefined behavior stop here?

There is no compendium of UB and IF-NDR in the standard. There is a project afoot to produce a list of UB for the core part of the language [5], but that project is not complete. So, for now, we can't easily determine how much UB and IF-NDR is present. But here's a quick sampling of undefined behavior that an intermediate programmer could easily stumble over:

The result of **accessing a non-common-initial sequence and non-active member of a union** is undefined. This situation is described in DIS 14882:2020 section 6.7.3 Lifetime [basics.life] paragraph 1:

> ... The lifetime of an object of type T begins when:
>
> — storage with the proper alignment and size for type T is obtained, and
>
> — its initialization (if any) is complete (including vacuous initialization) (9.4)
>
> except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (9.4.1, 11.10.2), or as described in 11.5 and 11.4.4.2, and except as described in 20.10.10.1.

The result of **accessing an indeterminate value other than through `unsigned char*` or `std::byte*`** is undefined. DIS 14882:2020 section 6.7.4 Indeterminate values [basic.indet] paragraph 2 describes the situation. The section is too long to quote.

Any race condition results in undefined behavior. DIS 14882:2020 section 6.9.2.1 Data races [intro.races] paragraph 21 says:

> The execution of a program contains a *data race* if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described below. Any such data race results in undefined behavior.

An infinite loop with no side effects results in undefined behavior. This can be inferred from DIS 14882:2020 section 6.9.2.2 Forward progress [intro.progress] paragraph 1 which, in a non-normative note, says:

> [*Note:* This is intended to allow compiler transformations such as removal of empty loops, even when termination cannot be proven. —*end note*]

The result of **signed integer overflow** is undefined. This is inferred from a section on mathematical operations. DIS 14882:2020 section 7.1 Preamble [expr.pre] paragraph 4 says:

> If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

The result of **subtracting two pointers that are not from the same array object** is undefined. DIS 14882:2020 section 7.6.6 Additive operators [expr.add] paragraph 5 says:

> When two pointer expressions P and Q are subtracted...
>
> — If P and Q both evaluate to null pointer values, the result is 0.
>
> — Otherwise, if P and Q point to, respectively, array elements i and j of the same array object x, the expression P - Q has the value i - j.
>
> — Otherwise the behavior is undefined...

The result of **a shift that is negative or exceeds an integer's size** is undefined. DIS 14882:2020 section 7.6.7 Shift operators [expr.shift] paragraph 1 says:

> The shift operators << and >> group left to right...

> ... The behavior is undefined if the right operand is negative, or greater than or equal to the width of the promoted left operand.

Flowing off the end of most non-void returning functions is undefined. DIS 14882:2020 section 8.7.3 [stmt.return] paragraph 2 says:

> Flowing off the end of a constructor, a destructor, or a non-coroutine function with a *cv* void return type is equivalent to a return with no operand. Otherwise, flowing off the end of a function other than main (6.9.3.1) or a coroutine (9.5.4) results in undefined behavior.

The result of **modifying a non-mutable const value** is undefined. DIS 14882:2020 section 9.2.8.1 The *cv-qualifiers* [dcl.type.cv] paragraph 4 says:

> Any attempt to modify (7.6.19, 7.6.1.5, 7.6.2.2) a const object (6.7.3) during its lifetime (6.6.3) results in undefined behavior.

In general, **user specialization of a function template defined in namespace std** is undefined behavior. This is specified by DIS 14882:2020 section 16.5.4.2.1 Namespace std [namespace.std] paragraph 1 which says,

> Unless otherwise specified, the behavior of a C++ program is undefined if it adds declarations or definitions to namespace std or to a namespace within namespace std.

Are we close to the end? No. There are approximately 195 direct references to undefined behavior in DIS 14882:2020. Sometimes those references are to identical sources of undefined behavior. But sometimes a single reference to undefined behavior introduces a list of multiple sources. And not every occurrence of undefined behavior is associated with the word "undefined."

In the library portion of DIS 14882:2020 section 16.5.4.11 Expects paragraph [res.on.expects] paragraph 1 says:

> Violation of any preconditions specified in a function's *Preconditions:* element results in undefined behavior.

So every instance of *Preconditions:* in the library portion of the standard declares some kind of undefined behavior.

There are 620 instances of *Preconditions:* in the library section of DIS 14882:2020.

So, between

- 195 direct references to undefined behavior, and
- 620 uses of *Preconditions:*,

we've now identified that there may be on the order of 815 instances of overt undefined behavior mentioned in DIS 14882:2020. And anywhere that the standard does not define *any* behavior is also undefined behavior. Instances of this kind of implicit undefined behavior are not easy to count. But it does mean that the 815 overt instances that we've counted are likely a lower bound to all of the undefined behavior in the standard.

There may be individuals who argue that undefined behavior in the library section of the standard is qualitatively different from undefined behavior in the core section of the standard. Some people refer to them as "soft" vs "hard" undefined behavior.

The standard itself makes no such distinction. As far as the standard is concerned any undefined behavior is just that: undefined. Claiming that there is a distinction between library and language UB is dangerous from a program security perspective. Many of the instances of UB in the library section of the standard result from efficient implementations that can, if misused, run afoul of core undefined behavior. For example, the effect of calling `front()` or `back()` for a zero-sized `std::array` is undefined. This is an efficiency trade-off to avoid checking the `array` size in every call to `front()` or `back()`. But violating the assumption may result in serious core undefined behavior.

There may be any number of instances of undefined behavior that an optimizer has not yet taken advantage of. But there is currently nothing in the standard that would prevent an optimizer (or library implementation) from taking advantage of undefined behavior in the library. In fact, given that STL interfaces are designed to be easily inlined, the optimizer already has visibility of many cases of core undefined behavior that can result from misuse of the standard library.

Therefore we should not dismiss the 620 identified cases of undefined behavior in the library section of the standard. Most of those may be "soft" undefined behavior for now. But there is no guarantee that they will remain so for any given implementation.

Ill-Formed No Diagnostic Required

Undefined behavior (UB) which is a runtime occurrence, has a close cousin in ill-formed no diagnostic required (IF-NDR). DIS 14882:2020 section 4.1 Implementation compliance [intro.compliance] paragraph 2.3 says,

> If a program contains a violation of a rule for which no diagnostic is required, this document places no requirement on implementations with respect to that program.

So IF-NDR is an attribute of an entire program. If the program is ill-formed and translation produces an executable the resulting program may do anything at all, much like undefined behavior. In those cases the programmer has created a program that could do any arbitrary thing, and the programmer probably has no visibility that they have a problem.

There are approximately 64 distinct instances of IF-NDR in DIS 14882:2020.

If we're going to consider "doing something" to improve the situation with UB, then we owe it to the users of the language to at least consider what can be done to reduce the occurrence of IF-NDR as well.

What Can Be Done?

Even if the standard were to provide a definitive list of UB and IF-NDR, such a list would not help intermediate programmers. There are too many instances in too many obscure corners. No intermediate programmer with a reasonable work-life balance will remember and recognize all the members of such a list.

It's also important to remember that, clearly, not all UB and IF-NDR situations can be removed from the standard. Some of them are too firmly entrenched in important optimization opportunities. Some of them *may* be removable, but the technique for each removal may be different from the others.

What we need is a process where knowledgeable people can propose changes to the standard that both:

- Improve the UB or IF-NDR situation and
- Are unlikely to have negative consequences.

What might that process look like?

A Process to Identify Useful Changes to the Standard

Suppose we agreed that it might be possible to tame some of the UB and IF-NDR in the C++ Standard. The techniques discussed in Appendix A might be useful. There may be other techniques that have not yet been identified. That leaves the larger question of how the specific changes would be identified and agreed upon. Here is one potential multi-phase approach.

1. The process will take a while; there's a lot to look at. In order to work from a stable base, the process would anchor itself to the C++20 DIS. Final conclusions from the process would then need to be forward ported into a future working draft.

2. The process would look at both UB and IF-NDR.

3. Three small teams would be identified:

    A. One or two people would build a spreadsheet of instances of UB and IF-NDR in the C++20 DIS. This team would also have responsibility for maintaining the spreadsheet throughout the effort. This team can coordinate their efforts with the people working on P1705 Enumerating Core Undefined Behavior [5].

    B. A small team of compiler internals experts and at least one wordsmith would focus on the core part of the standard. This team would be responsible for determining whether each specific instance of UB or IF-NDR could be reduced to something easier for a programmer to reason about.

    C. A different small team of standard library implementation experts and at least one wordsmith would focus on the library portion of the standard. This team would be responsible for determining whether each specific instance of UB or IF-NDR could be reduced to something easier for a programmer to reason about.

4. Team A would create a spreadsheet listing all overt cases of undefined behavior and ill-formed no diagnostic required situations. The format of the spreadsheet would be negotiated with the other two teams. At a minimum the spreadsheet would contain for each UB or IF-NDR instance:

    a. The page number (which will be stable since we're locking to the C++20 DIS),
    b. Section and paragraph, and
    c. A short description of the situation, to make the spreadsheet easier to read.

5. Once the initial list is built, additional non-explicit undefined behavior can also be incorporated into the spreadsheet. The two teams of experts would help to identify any non-explicit undefined behavior. Since non-explicit undefined behavior is harder to locate in the standard, it is anticipated that the final list will be incomplete.

6. Once the initial spreadsheet is filled in, team A will examine it and attempt to create sets of UB and IF-NDR that may be treated identically. For example, all instances of user specializations of primary templates defined in namespace `std` could be handled as one case. Similarly, all instances of `p is a valid iterator on *this` can be handled as a single case. This coalescing is done in an effort to reduce the work load on the two teams of experts.

7. Starting from that list, the teams of experts will audit each identified instance of UB and IF-NDR to determine whether, and if so how, any of it can be reduced to something easier for a programmer to reason about. Since these are experts, they will consider many aspects of each instance including impact on optimizations. The method for reduction would be up to the expert team members and would vary from instance to instance. Methods may include the ones described in Appendix A. Additional, not yet identified, methods may also be used. For each form of UB or IF-NDR one of three results is expected:

    a. The UB or IF-NDR is correctly labeled. No change suggested.

    b. The UB or IF-NDR could trivially be changed to another (specific) more desirable behavior with no negative impact. Or the scope of the UB or IF-NDR could be trivially reduced. A short justification for the change would be included along with proposed standard wording changes and mention of any risks.

    c. The UB or IF-NDR could be changed to another (specific) more desirable behavior with little or no negative impact given some non-trivial work. An outline of that non-trivial work would be included along with mention of any possible negative impact the change might have.

8. When the experts arrive at consensus about their conclusions, then the outcome is presented to the working groups including recommendations for changes if any.

    a. Any UB and IF-NDR that could be trivially changed to another specific behavior or have its scope reduced would be incorporated into an omnibus paper and sent through the committee, starting with SG12.

    b. Any UB or IF-NDR that requires non-trivial work to change would require additional, hopefully small, papers and further committee work.

Proposal Checklist

The following questions have been suggested as a proposal checklist [6].

What is the problem to be solved? Reducing the likelihood that intermediate C++ programmers stumble into undefined behavior or write ill-formed no diagnostic required code.

What kinds of users will be served? Intermediate C++ programmers.

What are alternative solutions? 1) Improve undefined behavior and ill-formed no diagnostic required detection with static analyzers and 2) improve education about UB and IF-NDR. None of the alternative solutions preclude the approach proposed in the paper. All approaches used in concert would further improve the usability of C++.

Why does the solution need to be in the standard? If the changes are not standardized then different implementation's optimizers and libraries will lead to different results.

What are the barriers to adoption? It is possible that turning some cases of UB or IF-NDR into ill-formed code would cause some programs to no longer compile. Different users will consider this either a blessing or a curse.

Has it been implemented? No, however the C++ Standards Committee has in the past been amenable to creating subgroups that build recommendations regarding specific topics of interest.

Will there be significant compile-time or run-time overhead? None is expected. Any possible changes that would introduce noticeable overhead would certainly be rejected by the small team of experts or by the committee.

Does the feature fit into the framework of existing tools and compilers? Yes. However it may have ripple effects with static analyzers and tools like UBSan.

Will there be compatibility problems? Potentially. See barriers to adoption.

Is the solution teachable? One goal of this proposal is to make C++ more teachable by reducing pitfalls. If this approach is successful then there will be (slightly) less to teach.

How will the standard library be affected? There is the potential for small changes to the standard library which improve compile-time checking or make small (backwards compatible) changes to prerequisites.

Will the proposal lead to demands for further extension in future standards? Not in any obvious fashion.

What mistakes are users likely to make with the new feature? If the proposal is successful there will be a slight reduction in mistakes that can be made with the C++ language.

Is the proposal for a general mechanism to solve a class of problems, or a specific solution to a specific problem? If to a class, which class of problems? The proposed approach is intended to address the broad class of instances of UB and IF-NDR in the standard.

Is the proposal coherent with the rest of the language in terms of semantics, syntax, and naming? Yes.

Summary

This paper argues that undefined behavior (UB) and ill-formed no diagnostic required (IF-NDR) situations make it difficult for an intermediate C++ programmer to reason about their program.

The paper proposes a process whereby much of the UB and all of the IF-NDR in the C++20 Standard would be audited by a small team of experts. The point of the audit would be to identify situations that are currently UB or IF-NDR and could potentially be converted to some more benign behavior. Each identified potential change would include an outline for how the change could be made and list any possible associated risks.

Thanks and Gratitude

The author would like to offer thanks to the plenitude of people who contributed to (but may not endorse) this paper. Specific important contributions came from: Aaron Ballman, Joshua Berne, Botond Ballo, J. F. Bastien, Walter Brown, Marc Glisse, Davis Herring, Howard Hinnant, Ryan Ingram, Erich Keane, Jens Maurer, John McFarlane, Melissa Mears, Robert Ramey, Gabriel Dos Reis, Hubert Tong, Ville Voutilainen, Jonathan Wakely, and JC van Winkle. That list is not complete. My thanks to you all, identified or not. All mistakes are the sole property of the author.

Appendix A: Speculative Considerations for the Audit

What kinds of changes could be made to the standard to reduce the amount of UB and IF-NDR? To get a handle on that we need to look at all of the currently available "behavior" options for a C++ program.

Available States/Behaviors for a C++ Program

The standard recognizes two fundamental states that a C++ program can be in. Those states are:

1. **well-formed program** (DIS 14882:2020 section 3.32 [defns.well.formed]). A C++ program constructed according to the syntax rules, diagnosable semantic rules, and the one-definition rule.
2. **ill-formed program** (DIS 14882:2020 section 3.12 [defns.ill.formed]) is a program that is not well formed. There are two kinds of ill-formed programs:
    a. **ill-formed no diagnostic required** is an ill-formed program that may or may not complete translation. If translation does complete the standard offers no guarantees for how the program runs.
    b. **ill-formed program that is guaranteed by the standard to produce a diagnostic**. Such a program may still complete translation and issue a warning. If translation completes then the standard offers no guarantees for how the program runs.

All C++ programs fit into one of these three categories, although it may be difficult for a programmer to know whether their program is well-formed or ill-formed no diagnostic required.

If the program is ill-formed, then the standard has little else to say about the program. The program may not complete translation. If it does complete translation then the program may do anything—it may crash, return erroneous results, or perform exactly as the programmer hoped.

If the program is well formed, then there are four kinds of behaviors that various places within the program may exhibit:

1. **defined behavior** is behavior that is specifically permitted and described by the standard.
2. **unspecified behavior** (DIS 14882:2020 section 3.31 [defns.unspecified]), which is still well-formed, but depends on the implementation. Therefore unspecified behavior is not portable.

3. **implementation-defined behavior** (DIS 14882:2020 section 3.13 [defns.impl.defined]) which is usually well formed, depends on the implementation, and is required to be documented by the implementation. Implementation defined behavior occasionally leads to implementation-specific undefined behavior. For example, an implementation is allowed to introduce data races (DIS 14882:2020 section 26.6.9 Low-quality random number generation [c.math.rand] paragraph 3) or specify which functions in the C++ standard library may be recursively reentered (DIS 14882:2020 section 19.5.5.9 Reentrancy [reentrancy] paragraph 1).
4. **undefined behavior** (DIS 14882:2020 section 3.30 [defns.undefined]) is behavior for which the standard imposes no requirements.

Notice that the preceding available behaviors for a program are run-time characteristics of that program. A program may, or may not, exhibit undefined behavior based on its inputs. Some inputs may result in undefined behavior and other inputs may not.

When committee members describe a construct in the standard, they may have more than one choice for which behavior describes it. In the next sections we'll explore the option of using choices other than UB and IF-NDR more often in the standard.

Please note that these examples are speculative. A UB expert might identify any number of reasons why these possible changes would be inappropriate. The following examples are also not intended to be comprehensive. The author suspects there are more examples to be found simply by looking a little deeper.

Preferring Unspecified Over Undefined Behavior

A poster child for this approach is DIS 14882:2020 section 27 Time library [time]. Due to the concerted effort of that section's primary author, there is only one explicit occurrence of undefined behavior in that entire section. Section 27.4.4 Class template is_clock [time.traits.is.clock] paragraph 2 says,

> The behavior of a program that adds specializations for `is_clock` is undefined.

Beyond that there are 9 *Preconditions:* specifications in that section, for a total of 10 instances of undefined behavior in that 90 page section of the standard. In contrast, there are 19 instances of unspecified behavior in the same section. This low undefined behavior count is because the primary author made a conscious decision to prefer unspecified behavior over undefined behavior.

So it's important to remember when wording the standard that there are choices other than undefined behavior when a specific construct is not precisely defined. Simply by making other choices, such as unspecified or implementation-defined behavior, when adding new features to the specification, or when editing it, it is possible to reduce the amount of undefined behavior in C++.

Making Selected UB or IF-NDR Ill-Formed

First, why might it be preferable for a program to be ill-formed rather than contain undefined behavior? A program that is ill-formed (usually) fails to compile and produces a diagnostic. This allows the programmer to see that there is a problem and identify ways to correct it. Undefined behavior, on the other hand, may simply allow the program to run and produce unexpected results. Which would you choose?

On the other hand, there are certainly existing programs that currently contain UB or IF-NDR code and work well enough with a specific compiler and with specific compile switches to suit their purpose. If the UB or IF-NDR code in those programs becomes ill-formed, then users would be forced to fix such programs. Some people might object to this sort of backwards incompatibility, even though the code has UB or is IF-NDR. They might object strenuously if the program has UB that is never exercised.

Backwards incompatibility is a legitimate concern. Addressing that concern would need to be done on a case-by-case basis.

User Specializations of Certain Templates in Namespace `std`

There are quite a number of cases of undefined behavior in the library section of the standard when a user specializes a template defined in namespace `std`. Just a few examples include:

- 17.11.5 Result of three-way comparison [cmp.result] paragraph 1
- 17.12.3 Class template coroutine_handle [coroutine.handle] paragraph 2
- 20.15.1 Requirements [meta.rqmts] paragraph 4
- 20.18.3 Execution policy type trait [execpol.type] paragraph 3
- 27.4.4 Class template is_clock [time.traits.is.clock] paragraph 2

The committee might consider defining some form of decoration, possibly an attribute, that could optionally be applied to primary templates such that specializing the template is ill-formed. Once such a decoration existed then it could be applied within the standard library. That could potentially make many of these cases of undefined behavior ill-formed instead. Programmers would be told, at compile time, that they had written bad code.

A change like this would only provide incomplete coverage. Only user specializations of templates in `std` that:

- require *no* specializations (by the standard library) within `std` itself and
- disallow *all* specializations by users

could be made ill-formed. The committee would have to decide whether such partial coverage was worth the effort. The audit could help inform that decision, since it would show the number of cases that changed from UB to ill-formed. The committee would also need to decide whether inconsistent enforcement is better than no enforcement at all.

Such a change would also be possible without standardizing a new decoration. If the standard said that a user specializing a template defined in namespace `std` must be diagnosed, then each implementation could define its own decoration.

The Preprocessor and UB and IF-NDR

There are a few places where the preprocessor stumbles into UB or IF-NDR cases. For example, DIS 14882:2020 section 5.7 Comments [lex.comment] paragraph 1 says that in certain circumstances a form-feed or vertical-tab character in a comment is IF-NDR. Have our parsers now become good enough that some of the preprocessing concerns can simply be ill-formed? Just wondering…

Flowing Off the End of a Non-`void` Returning Function

DIS 14882:2020 section 8.7.3 [stmt.return] paragraph 2 says flowing off the end of most non-`void` returning functions is undefined behavior. Usually this is a programming error that is easily detected by the compiler. And it is a very easy mistake for a programmer to make. However there are situations where the programmer knows that a function will never flow off the end, but it looks to the compiler like it might happen. Should we let these few situations make the language a more dangerous place for all users?

One approach to this problem would be to require programmers to mark situations where flowing off the end of a function is programmatically not possible. The mark would need to be a new decoration of some sort. Using a call to a preexisting construct such as `std::abort()`, or any other function with a `[[noreturn]]` attribute, as such a mark would cause the compiler to insert code that an optimizer might not be able to remove. But, as long as the source code is available and modifiable, then placing the new decoration at the end of the non-`void` returning function solves it.

Some compilers provide extensions to inform the compiler that a certain code path (such as flowing off the end of a non-void function) is unreachable, e.g `__builtin_unreachable`. That could be standardized. Or the `[[noreturn]]` attribute could be reused. Instead of the more general `__builtin_unreachable` (which can appear anywhere in a function) the `[[noreturn]]` attribute could be allowed on an empty statement at the end of a function, to indicate that omitting the return statement is intentional because it isn't reachable.

It doesn't eliminate the potential for undefined behavior, because the user who adds that decoration could be wrong, and if control flow does actually get there it's still undefined. But the common case of accidentally forgetting the return statement or failing to handle a conditional case could be turned into a required diagnostic. If the user explicitly adds `[[unreachable]]` or `[[noreturn]]` we've reduced the incidence of UB to the cases where they're explicitly wrong, not just accidentally careless.

There are, admittedly, at least two reasons that this possible change could be rejected by the committee:

1. Pre-existing code that flows off the end of a non-void function would become malformed if no source code modifications were made. That breaks backward compatibility. That's a legitimate concern which would require discussion.
2. There are members of the committee that are clever enough that they would never make a mistake like this. They may object to the required extra text in their programs. To those members of the committee the author suggests that they can consider a requirement like this to be similar to wearing a mask during the COVID-19 pandemic. You would be making a sacrifice to increase the safety of a more vulnerable part of the population.

The One Definition Rule

DIS 14882:2020 section 6.3 One-definition rule [basic.def.odr] paragraph 10 says

> Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program outside of a discarded statement (8.5.1); no diagnostic required.

Consider that we now live in a world with modules. In certain situations with modules the translation process may have visibility of the entire program. In those situations could violating the ODR rule become simply ill-formed, rather than IF-NDR?

Making Selected Undefined Behavior Well-Formed

It's possible that certain undefined behavior could be made well-formed.

memcpy

Consider memcpy. Passing a nullptr or other unusable pointer to memcpy results in undefined behavior, even if the length of the copy is zero. DIS 14882:2020 doesn't say much about memcpy, but the C11 Standard identifies this undefined behavior [8]. C11 Standard section 7.24.1 String function conventions paragraph 2 states:

> Where an argument declared as size_t n specifies the length of the array for a function, n can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid values, as described in 7.1.4.

Then C11 Standard section 7.1.4 Use of library functions paragraph 1 states:

> If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer, or a pointer to non-modifiable storage when the corresponding parameter is not const-qualified) or a type (after promotion) not expected by a function with variable number of arguments, the behavior is undefined.

Well, C++ is not the same language as C. If we wanted to make memcpy considerably easier to use, then C++ could specify that if the length of a memcpy is zero then the pointer values are irrelevant. And that little bit of undefined behavior would be eliminated from the language.

There would certainly be complexities. The C++ standard requires ::memcpy and std::memcpy to be the same. DIS 14882:2020 section 21.5.3 Header <cstring> synopsis [cstring.syn] paragraph 1 says

> The contents and meaning of the header <cstring> are the same as the C standard library header <string.h>.

So if this change were made, every C++ implementation would be forcing a particular memcpy implementation on the associated C library. This could potentially be done in concert with the C Standard Committee. Or, since the C++ implementation would also be compatible with the C

standard, we could simply let each compiler/library implementation solve the problem however it sees fit.

Infinite Loops

Also consider infinite loops with no side effects (DIS 14882:2020 section 6.9.2.2 Forward progress [intro.progress] paragraph 1). This paragraph states the assumption that any thread will terminate. That's a poor assumption for quite a number of existing embedded programs that run until the power is removed or there is a (not visible to the program) hardware reset.

To further expand on the usefulness of infinite loops, not all embedded environments have easy access to traditional I/O, or even breakpoints. When such a platform runs into a truly unexpected situation an easy debugging technique is to simply go into an infinite loop at the address where the problem is discovered. The embedded system hardware can then be probed externally, for example using a logic analyzer, to discover the address where the failure was detected. The author has worked on such deeply embedded systems and used the infinite loop technique to great effect.

If the optimizer removes such a loop then the person who inserted the loop will be left puzzling why the system crashed without getting stuck in the loop. Vast confusion ensues.

It's worth pointing out that the debugging loop can be made un-removable in the current state of the standard by putting a `volatile` access inside the loop. And an expert would make such a transformation. An intermediate programmer might never figure out that such a work-around was required or even exists.

Also the compiler may not be able to determine whether a given loop terminates. In that case the solution is simple; if the optimizer can't figure out whether the loop terminates, then it should leave the loop in place. The optimizer would be allowed to remove loops that it can prove terminate.

An optimizer implementer might indicate that this change would have serious implications for optimizations on the many platforms where infinite loops don't make sense. That would be a legitimate concern. But it would still leave open the option of making the behavior of infinite loops implementation-defined. Or perhaps it could be conditional on freestanding/hosted implementations.

In summary, it might make sense for infinite loops with no side effects to be well-formed, at least on freestanding implementations. Certainly, whether or not a thread (or program) terminates could easily be considered observable behavior.

Refine Proscription Wording Within the Standard

Consider DIS 14882:2020 section 16.5.4.2.1 Namespace std [namespace.std] paragraph 6:

> Let *F* denote a standard library function (16.5.5.4), a standard library static member function, or an instantiation of a standard library function template. Unless *F* is designated an *addressable function*, the behavior of a C++ program is unspecified (possibly ill-formed) if it explicitly or implicitly attempts to form a pointer to *F*.

As this paragraph stands, the behavior of a program is unspecified if a programmer takes the address of a standard library function. Possibly the paragraph could be rephrased to be less dangerous. Consider…

> … Unless *F* is designated an *addressable function*, the result of explicitly or implicitly forming a pointer to *F* has an unspecified type and value; it is also unspecified whether a program that forms such a pointer is ill-formed.

With the current phrasing of the standard, the behavior of a program that simply forms a pointer to *F* is unspecified. With this possible rewording the unspecified behavior is postponed until the pointer is actually used (by using a pointer of unspecified type and value). And an implementation is still allowed to notice the creation of such a pointer and declare the program ill-formed.

Wherever such a refinement in wording can be applied, and this instance is only one example, it has the potential to reduce the surface area of UB and IF-NDR within the standard.

Consider Adding a New Term of Art

It is possible that some forms of UB or IF-NDR could be transformed by introducing a new term of art to the standard. For example the Core working group has bandied about a term of art (tentatively "unspeciformed") to describe a construct that is exactly one of two things:

- Ill-formed in a way that can be detected by the compiler and linker, or
- If the code is not detected as ill-formed, the code is well-formed.

It's possible that this approach would help remove some kinds of IF-NDR. Here's how Joshua Berne has described a potential use case:

> Consider the case where there are zero definitions of a one-definition-rule-used function. This case is currently IF-NDR. But in practice you only get two possible outcomes:
>
> 1. Either all references to the function get optimized away so your program links and behaves correctly, or
>
> 2. You have references to the function and your program fails to link.
>
> Currently, I believe a compliant compiler is allowed to link such a program and give you a chicken instead. Making this unspeciformed instead would better reflect existing practice and make the world a safer place.

Of course, as we all know, a compiler is very unlikely to produce a chicken.

However we would like users of the language to take IF-NDR as a very serious concern, right? Link errors are probably among the most common form of IF-NDR. If these sorts of link errors never, in fact, produce a program that behaves in an ill-formed way then we should not be giving them a scary name. We should describe them as they are—ill-formed if an error is produced and otherwise well-formed. We should reserve the scary name for situations that should actually scare our users. Otherwise we are teaching them to ignore the scary name.

There's a chance that such a change for link errors could be made in the standard without introducing a new term of art. So this specific case may not be the place that requires a new term of art.

Still, if the audit is undertaken, staying open to adding one or more new terms of art may be an important way to manage some cases of UB or IF-NDR. We should keep an open mind.

References

[1] *C++20 Draft International Standard ISO/IEC DIS 14882:2020*. Voting began on June 11, 2020.

[2] Schurr, Scott. *P1407R1. Tell Programmers About Signed Integer Overflow Behavior*. March 8, 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1407r1.pdf

[3] van Winkel et al. *P0559R0. Operating Principles for Evolving C++*. January 31, 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0559r0.pdf

[4] Hinnant et al. *P2000R1. Direction for ISO C++*. January 13, 2020. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2000r1.pdf

[5] Yaghmour, Shafik. *P1705R1. Enumerating Core Undefined Behavior*. September 29, 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1705r1.html

[6] Stroustrup, Bjarne. *Thriving in a crowded and changing world: C++ 2006-2020*. Proceedings of the ACM on Programming Languages, June 2020, article No.: 70, pp. 70:28-29. https://dl.acm.org/doi/abs/10.1145/3386320.

[7] Kahneman, Daniel. *Thinking, Fast and Slow*. Farrar, Straus, and Giroux, 2011, pp. 283 - 286.

[8] Committee Draft N1570. *ISO/IEC 9899:201x Programming Languages — C.* April 12, 2011.