The difference between undefined behavior and ill-formed C++ programs

devblogs.microsoft.com/oldnewthing/20240802-00

August 2, 2024

Raymond Chen

The C++ language has two large categories of “don’t do that” known as *undefined behavior* and *ill-formed program*. What’s the difference?

Undefined behavior (commonly abbreviated *UB*) is a runtime concept. If a program does something which the language specified as “a program isn’t allowed to do that”, then the behavior at runtime is undefined: The program is permitted by the standard to do anything it wants. Furthermore, the effect of undefined behavior can go backward in time and invalidate operations that occurred prior to the undefined behavior. It can do things like execute dead code. However, if your program avoids the code paths which trigger undefined behavior, then you are safe.

```
int nervous(bool is_scary, int n)
{
    if (is_scary) {
        return 100 / n;
    } else {
        return 0;
    }
}

int main()
{
    return nervous(false, 0);
}
```

There is no undefined behavior in this program because `nervous` is called with `is_scary` set to `false`, so the `return 100 / n` never executes, and we avoid the division by zero.

Avoiding code paths with undefined behavior is something you do all the time.

```
// Check the pointer before using it
if (p != nullptr) p->DoSomething();

// Avoid division by zero
return (n == 0) ? 0 : 100 / n;

int a[5]{};
// Check array index before using it
if (n < 5) return a[n];
```

Even if a program contains undefined behavior, the compiler is still obligated to produce a runnable program. And if the code path containing undefined behavior is not executed, then the program's behavior is still constrained by the standard.

By comparison, an *ill-formed* program is a program that breaks one of the rules for how programs are written. For example, maybe you try to modify a variable declared as `const`, or maybe you called a function that returns `void` and tried to store the result into a variable.

There are two subcategories of ill-formed programs. One is the plain vanilla ill-formed program, for which the standard requires a *diagnostic*, meaning that the standard requires that the compiler report the error, and the compiler is no longer obligated to produce a runnable program. Indeed, by default, most compilers will refuse to produce *any* program at all, runnable or not.

```
const int size = 4;

void expand()
{
    // ill-formed: Modifying a const variable.
    size = 9;
}
```

The above program is ill-formed because it tries to modify a const variable, and the compiler reports a compile-time error and refuses to produce a program. The plain vanilla ill-formed programs aren't scary because the compiler lets you know that you broke a rule and typically refuses to let you proceed until you fix it.

The other subcategory of ill-formed programs is the scary one: ill-formed no diagnostic required, commonly abbreviated *IFNDR*. These are programs which are ill-formed, but for which the standard does not require the compiler to report an error. The compiler is welcome to do so if it chooses, but it is also permitted to remain silent. In practice, IFNDR is used to describe things which are “bad” but which compilers are not equipped to detect. For example, if you have two translation units (standard-speak for “a .cpp file”), both of them must agree on the bodies of any inline functions.

```
// file1.cpp

inline int magic() { return 42; }
int get_value() { return magic(); }

// file2.cpp

extern int get_value();
inline int magic() { return 99; }

int main(int argc, char** argv)
{
    if (argc > 1) return get_value();
    return 0;
}
```

In the above example, we have a project that consists of two .cpp files. The two files disagree on what the `int magic()` inline function does, which is a category of IFNDR. The compiler is permitted but not required to detect this mismatch, and if you run the resulting program, the results are undefined: If you run the resulting program with a command line argument, the `get_value()` function might return 42. It might return 99. It might return 31415. It might reformat your hard drive. It might hang.

Even worse, even if you run the program with no command line options (so that `get_value()` is never called), the results are *still* undefined. It could still reformat your hard drive.

That’s what’s scary about IFNDR: the resulting program is *already broken even before you run it*. If a program contains IFNDR, the standard imposes no requirements on the behavior when you run the resulting program. The program is *fundamentally invalid*, and no good will come out of running it.

In practice, IFNDR causes your program to behave erratically: In the above example, when you try to debug the program, you may observe that calls to `magic()` return values that don’t make sense because the compiler chose to use a copy of the inline function that is different from the one you expected. If your IFNDR results from an inconsistent class declaration, you may experience memory corruption because two different parts of the program disagree on what the class layout is.

A common source of IFNDR is making changes to a class dependent upon a preprocessor setting that is different in different .cpp files.

```
// common.h

struct Widget
{
    Widget();

    ⟦ more stuff ⟧

#ifdef EXTRA_WIDGET_DEBUGGING
    Logger m_logger;

    void Log(std::string const& message) {
        m_logger.log(message);
    }
#else
    void Log(std::string const& message) {
        // no extra logging
    }
#endif

    std::string m_name;
};
```

If two .cpp files include this common header file, and one of them defines `EXTRA_WIDGET_DEBUGGING` but the other does not, then you have a big problem because they will disagree on `sizeof(Widget)`, they will disagree on where the `m_name` is, and they will disagree on what the `Log()` function does. The result is that any use of the `Widget` will choose one definition or the other, and if not everybody chooses consistently (and in practice, there's a good chance they won't), then you have memory corruption on your hands.

Visual Studio has an unofficial command line option to help identify certain classes of IFNDR. And you can code defensively and use `#pragma detect_mismatch` to help catch these types of mismatches.