# How to check if a real number is an integer in C++?

2022-07-18 ()

I have a `double`, and I want to know if its value is an integer that fits in a `int64_t`. How can I do it in C++?

Ask any C++ newbie, and you will get an obvious "answer": cast your `double` to `int64_t`, then cast it back to `double`, and compare if it equals your original number.

```
bool IsInt64(double d) {
    return d == static_cast<double>(static_cast<int64_t>(d));
}
```

But is it really correct? Let's test it:

```
int main() {
    printf("%s", IsInt64(1e100) ? "1" : "0");
    return 0;
}
```

and here's the output (https://godbolt.org/z/r8j98rKqd) under `clang -O3` (latest version 14.0.0):

```
▯n▯▯▯
```

(https://sillycross.github.io/images/2022-07-18/output1.png)

(a bunch of junk characters that varies from run to run)

!@#$%^&... Why? Shouldn't it at least print either a `1` or a `0`?

## The Undefined Behavior

Here's the reason: when you cast a floating-point value to an integer type, according to C/C++ standard (https://en.cppreference.com/w/cpp/language/implicit_conversion#Floating.E2.80.93integral_conversions), if the integral part of the value does not fit into the integer type, the behavior is undefined (by the way, casting special floating-point values `NaN`, `INF`, `-INF` to integer is also undefined behavior).

And unfortunately, Clang did the least helpful thing in this case:

1. It inlined the function `IsInt64`, so `IsInt64(1e100)` becomes expression `1e100 == (double)(int64_t)1e100`.
2. It deduces that `(int64_t)1e100` incurs undefined behavior since `1e100` does not fit into `int64_t`, so it evaluates to a special `poison` value (i.e., undefined).
3. Any expression on a `poison` value also produces `poison`. So Clang deduces that expression `IsInt64(1e100) ? "1" : "0"` ultimately evaluates to `posion`.
4. As a result, Clang deduces that the second parameter to `printf` is an undefined value. So in machine code, the whole expression is "optimized out", and whatever junk stored in that register gets passed to `printf`. `printf` will interpret that junk value as a pointer and prints out whatever content at that address, yielding the junk output.

Note that even though `gcc` happens to produce the expected output in this case, the undefined behavior is still there (as all C/C++ compilers conform to the same C/C++ Standard), so there is no guarantee that the `IsInt64` function above will work on `gcc` or any compiler.

So how to implement this innocent function in a standard-compliant way?

## The Bad Fix Attempt #1

To avoid the undefined behavior, we must check that the `double` fits in the range of the `int64_t` before doing the casting. However, there's a few tricky problems involved:

1. While `-2^63` (the smallest `int64_t`) has an exact representation in `double`, `2^63-1` (the largest `int64_t`) doesn't. So we must be careful about the rounding problems when doing the comparison.

2. Comparing the special floating-point value `NaN` with any number will yield `false`, so we must write our check in a way that `NaN` won't pass the check.
3. There is another weird thing called negative zero (https://en.wikipedia.org/wiki/Signed_zero) (`-0`). For the purpose of this post, we treat `-0` same as `0`. If not, you will need another special check.

With these in mind, here's the updated version:

```
bool IsInt64(double d) {
  if (-9223372036854775808.0 <= d && d < 9223372036854775808.0) {
    return d == static_cast<double>(static_cast<int64_t>(d));
  } else {
    return false;
  }
}
```

However, unfortunately, while the above version is correct, it results in some unnecessarily terrible code on x86-64:

```
.LCPI0_0:
  .quad   0xc3e0000000000000     # double -9.2233720368547758E+18
.LCPI0_1:
  .quad   0x43e0000000000000     # double 9.2233720368547758E+18
IsInt64(double):                              # @IsInt64(double)
  xor        eax,  eax
  ucomisd    xmm0, qword ptr [rip + .LCPI0_0]
  jb         .LBB0_3
  movsd      xmm1, qword ptr [rip + .LCPI0_1]
  ucomisd    xmm1, xmm0
  jbe        .LBB0_3
  cvttsd2si  rax,  xmm0
  xorps      xmm1, xmm1
  cvtsi2sd   xmm1, rax
  cmpeqsd    xmm1, xmm0
  movq       rax,  xmm1
  and        eax,  1
.LBB0_3:
  ret
```

In fact, despite that out-of-range floating-point-to-integer cast is undefined behavior in C/C++, the x86-64 instruction `cvttsd2si` used above to perform the cast is well-defined on all inputs (https://www.felixcloutier.com/x86/cvttsd2si): if the input doesn't fit in `int64_t`, then the output is `0x80000000 00000000`. And since `0x80000000 00000000` has an exact representation in `double`, casting it back to `double` will yield `-2^63`, which won't compare equal to any `double` value but `-2^63`.

So the range-check is actually unnecessary for the code to behave correctly on x86-64: it is only there to keep the C++ compiler happy, but unfortunately, the C++ compiler is unable to realize that such check is unnecessary on x86-64, thus cannot optimize it out on x86-64.

To summarize, on x86-64, all we need to generate is the last few lines of the above code.

```
IsInt64(double):                              # @IsInt64(double)
  cvttsd2si  rax,  xmm0
  cvtsi2sd   xmm1, rax
  cmpeqsd    xmm1, xmm0
  movq       rax,  xmm1
  and        eax,  1
  ret
```

But is there any way we can teach the compiler to generate such assembly?

## The Bad Fix Attempt #2

In fact, our original buggy implementation

```
bool IsInt64(double d) {
    return d == static_cast<double>(static_cast<int64_t>(d));
}
```

produces exactly the above assembly (https://godbolt.org/z/66Y4nKc1b). The problem is, whenever the optimizer of the C++ compiler inlines this function and figures out that the input is a compile-time constant, it will do constant propagation according to C++ rule – and as a result, generate the `poison` value. So can we stop the optimizer from this unwanted optimization, while still having it doing optimizations properly for the rest of the program?

In fact, I have posted this question on LLVM forum months ago, and didn't get an answer. But recently I suddenly had an idea. `gcc` and `clang` all support a crazy builtin (https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html) named `__builtin_constant_p`. Basically this builtin takes one parameter, and returns `true` if the parameter can be proven by the compiler to be a compile-time constant[1]. Yes, the result of this function is dependent on the optimization level!

This builtin has a very good use case: to implement `constexpr offsetof`. If you are certain that some expression `p` is a compile-time constant, you can do `constexpr SomeType foo = __builtin_constant_p(p) ? p : p;` to forcefully make `p` a `constexpr`, even if `p` is not `constexpr` by C++ standard, and the compiler won't complain anything! This allows one to perform `constexpr reinterpret_cast` between `uintptr_t` and pointers, thus implement a `constexpr` -version `offsetof` operator.

However, what I realized is that, this builtin can also be used to prevent the unwanted constant propagation. Specifically, we will check `if (__builtin_constant_p(d))`. If yes, we run the slow-but-correct code – this doesn't matter as the optimizer is going to constant-fold the code anyway. If not, we execute the fast-but-UB-prone code, which is also fine because we already know the compiler can't constant-fold anything to trigger the undefined behavior.

The new version of the code is below:

```
// DON'T USE IT!
bool IsInt64(double d) {
  if (__builtin_constant_p(d)) {
    // Run UB-free version, knowing that it's going to
    // be constant-folded by the compiler any way
    if(-9223372036854775808.0 <= d && d < 9223372036854775808.0) {
      return d == static_cast<double>(static_cast<int64_t>(d));
    } else {
      return false;
    }
  } else {
    return d == static_cast<double>(static_cast<int64_t>(d));
  }
}
```

I tried the above code on a bunch of constants and non-constant cases, and the result seems good. Either the input is correctly constant-folded, or the good-version assembly is generated.

So I thought I outsmarted the compiler in this stupid Human-vs-Compiler game. But am I...?

## Don't Fight the Tool!

Why does C/C++ have this undefined behavior after all? Once I start to think about this problem, I begin to realize that something must be wrong...

The root reason that C/C++ Standard specifies that an out-of-range floating-point-to-integer cast is undefined behavior is because on different architectures, the instruction that performs the float-to-int cast exhibits different behavior when the floating-point value doesn't fit in the integer type. On x86-64, the behavior of the `cvttsd2si` instruction in such cases is to produce `0x80000000 00000000`, which is fine for our use case. But what about the other architectures?

As it turns out, on ARM64, the semantics of the `fcvtzs` instruction (analogue of x86-64's `cvttsd2si`) is saturation: if the floating-point value is larger than the max value of the integer type, the max value is produced; similarly, if the floating-point value is too small, the minimum integer value is produced. So if the `double` is larger than `2^63-1`, `fcvtzs` will produce `2^63-1`, not `-2^63` like in x86-64.

Now, recall that `2^63-1` doesn't have an exact representation in `double`. When `2^63-1` is cast to `double`, it becomes `2^63`. So if the input `double` value is `2^63`, casting it to `int64_t` (`fcvtzs x8, d0`) will yield `2^63-1`, and then casting it back to `double` (`scvtf d1, x8`) will yield `2^63` again. So on ARM64, our code will determine that the `double` value `2^63` fits in `int64_t`, despite that it actually does not.

I don't own a ARM64 machine like Apple M1, so I created a virtual machine using `QEMU` to validate this. Without surprise, on ARM64, our function fails when it is fed the input `2^63`.

So clearly, the undefined behavior *is* there for a reason...

## Pick the Right Tool Instead!

As it turns out, I really should not have tried to outsmart the compiler with weird tricks. If performance is not a concern, then the UB-free version is actually the only portable and correct version:

```
bool IsInt64(double d) {
  if (-9223372036854775808.0 <= d && d < 9223372036854775808.0) {
    return d == static_cast<double>(static_cast<int64_t>(d));
  } else {
    return false;
  }
}
```

And if performance *is* a concern, then it's better to simply resort to architecture-dependent inline assembly. Yes, now a different implementation is needed for every architecture, but at least it's better than dealing with hard-to-debug edge case failures.

Of course, the ideal solution is to improve the compiler, so that the portable version generates optimal code on every architecture. But given that neither `gcc` nor `clang` had supported this, I assume it's not an easy thing to do.

---

### Footnotes

1. Note that this builtin is different from the C++20 `std::is_constant_evaluated()`. The `is_constant_evaluated` only concerns whether a `constexpr` function is being evaluated constexpr-ly. However, `__builtin_constant_p` tells you whether a (maybe non-constexpr) expression can be deduced to a compile-time known constant under the current optimization level, so it has nothing to do with `constexpr`. ↩

---


Bizarre Performance Characteristics of Alder Lake CPU (../../../06/11/2022-06-11/)

Pitfalls of using C++ Global Variable Constructor as a Registration Mechanism (../../../10/02/2022-10-02/)