Philippe Dunski
dunski.philippe@gmail.com

Disambiguating overflow/underflow constexpr behaviour

Abstract

In this paper, I'll propose you to fix some inconsistant compiler's behaviour when dealing with constexpression, and specially, its behaviour face to (unsigned) integer overflow/underflow.

Motivation

C++ is known for its point of view that « developer knows what he is doing ».

It means that compiler should not go against the developer's decision, especially when developper makes use of constexpr.

From my point of view, it also means that, any intger constexpr's result should be constrainted by its underlying type range, regardless of wether this type is signed or unsigned.

The problem

On this time, compiler's behaviour facing to integers is defined by three main lines :

- conversion to another type (which is irrelevant here)
- promotion which can araise for any type smaller than (unsigned) long long and
- overflow rules, which specifically says that UINT_MAX + 1 == 0 == UINT_MIN -1 (is an undefined behaviours for signed integers)

At runtime, such main lines should not be a problem, but when dealing with constexpr, it is.

There are – at least – twoo good reasons to consider the runtime behaviour to be bad for constexpr.

Compilers behaviour inconsistancy

Please take a look at following code.

```
#include <type_traits>
#include <limits>
#include <cstdint>
using namespace std;

using I8Min = integral_constant<int8_t, numeric_limits<int8_t>::min()-1>;
using I16Min = integral_constant<int16_t, numeric_limits<int16_t>::min()-1>;
using I32Min = integral_constant<int32_t, numeric_limits<int32_t>::min()-1>;
using I64Min = integral_constant<int64_t, numeric_limits<int64_t>::min()-1>;

using I8Max = integral_constant<int8_t, numeric_limits<int8_t>::max()+1>;
using I16Max = integral_constant<int16_t, numeric_limits<int16_t>::max()+1>;
using I32Max = integral_constant<int32_t, numeric_limits<int32_t>::max()+1>;
using I64Max = integral_constant<int64_t, numeric_limits<int64_t>::max()+1>;

using UI8Min = integral_constant<uint8_t, numeric_limits<uint8_t>::min()-1>;
using UI16Min = integral_constant<uint16_t, numeric_limits<uint16_t>::min()-1>;
using UI32Min = integral_constant<uint32_t, numeric_limits<uint32_t>::min()-1>;
using UI64Min = integral_constant<uint64_t, numeric_limits<uint64_t>::min()-1>;
```

```
using UI8Max = integral_constant<uint8_t, numeric_limits<uint8_t>::max()+1>;
using UI16Max = integral_constant<uint16_t, numeric_limits<uint16_t>::max()+1>;
using UI32Max = integral_constant<uint32_t, numeric_limits<uint32_t>::max()+1>;
using UI64Max = integral_constant<uint64_t, numeric_limits<uint64_t>::max()+1>;
```

We have – basically – 16 overflow / underflow behaviours. We could expect to get exactely the same behaviour for every case. But, we don't :

- Some compilers will only consider signed values as compile time errors, but not unsigned values
- Some compilers will consider all unsigned values and only MIN-1 unsigned value as error
- Some compiler will only add uint8_t and uint16_t at them error list with some specific options
- Some compilers are just fine with this code and will only produce a warning if some specific option is set.

And, behind the presence of a compile time, I saw some compiler version giving two different reasons for those errors.

An unexpected « loop » in constexpr values

Even worst than compiler inconsistancy is the possible presence of an unexpected – and error prone – loop in constexpr value.

Please take a look at the following code :

```
template <typename IndexType, typename T, typename... Ts>
struct IndexImpl;

template <typename IndexType, typename T, typename... Ts>
struct index_impl<IndexType, T, T, Ts...> : integral_constant<IndexType, 0> {};

template <typename IndexType, typename T, typename U, typename... Ts>
struct index_impl<IndexType, T, U, Ts...> :
   integral_constant<IndexType, 1 + index_impl<IDX,T, Ts...>::value> {};
template <typename IndexType, typename ... TS>
using type_index = index_impl<IndexType, T, TS ...>::value;
```

It is, basically, a more or less naive way to get the index of some type in a variadic templated ist taking the form of

```
template <typename .. . Args>

struct type_list{} ;
```

Everything seems to be perfectly fine, isn't it ? But, now, consider to use such TypeIndex in a way like

```
using my_list = my_list</* 256 or more user defined types*/>;

constexpr uint8_t my_index = type_index<uint8_t, some_t, type_index>;
```

What 256th type's value will be computed to ... 0. But 0 is already the first type's index.

Surprizing, isn't it ?

Contra argumentation

One could argue that

> It's perfectly normal, since your uint8_t is promoted into uin16_t when computing max + 1, and since result doesn't overflow.

Another could argue that

> Changing this could cause an incompatiblity with C.

And I'm in peace with those arguments... when speaking about **runtime**.

But, here, I'm specifically speaking about constexpr. In other words, I'm speaking about compiler's constant. Promotion and possible incompatibility with C are just ... irrelevant here.

At this time, it's just like if we let the compiler to tell the developper something like

> I know that you asked some (u)intX_t value, but i'll give you an (u)int2X_t one because of the onder/underflow

My conviction is that it **should not append** : As far as the developer specifically asked for some well sized type, compiler **should** give him the requested type in respect for its allowed range.

If, some any reason, computed value does'nt fit in that range, compiler should have only one possible anwer : to fail at compile time, regardless wether the requested type is signed or unsigned.

And developer should be advised that he has choosen a too small type for this value.

Proposed wording

To fix this issue, some addition should be required in 7,6 [conv.prom] and in 8.6 [expr.const].Additionnaly some change should be done in 19.1.10 [cpp.cond] :

Addition in 7.6 [conv.prom]

An 8th subclause should be added to conv.prom. It could be simply worded in the form of

> No promotion should never arise on in any integer type when dealing (unsigned) integer constexpr. This interdiction is prioritary over any other consideration

(nota : as far as such interdiction is mandatory and should have precedence on any other consideration, we could maybe put it as very first subclause)

We could also make a little change in 7.8.1 (conv.integral)

> A prvalue of an integer type can be converted to a prvalue of another integer type. A prvalue of an unscoped enumeration type can be converted to a prvalue of an integer type **if and only if integer type isn't constexpr**.

Addition in 8.6 [expr.const] :

An 8th subclause should be added to 8.6[expr.const] in the forme of :

> integer constexpr should be consrainted to the underlying type's range without any opportunity to overflow or underflow. It is true and mandatory for signed and unsigned integers.

(note : as far as this rules should have precedence on any other consideration, its place could be reconsidered)

Modification to 19.1.10 [cpp.cond]

Finally, I'll suggest to make some modification in 19.1.10 [cpp.cond] to make thinks very clear :

> The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 8.6 using arithmetic that has at least the ranges specified in 21.3. For the purposes of this token conversion and evaluation all signed and unsigned integer types act as if they have the same representation as, respectively, intmax_t or uintmax_t (21.4). [Note: Thus on an implementation where std::numeric_limits<int>::max() is 0x7FFF and std::numeric_limits<unsigned int>::max() is 0xFFFF, the integer literal 0x8000 is signed and positive within a #if expression even though it is unsigned in translation phase 7 (5.2). **But, if such overflow occures with integer constepr, compilation will fail** —end note] This includes interpreting character literals, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character literals matches the value obtained when an identical character literal occurs in an expression (other than within a #if or #elif directive) is implementation-defined. [Note: Thus, the constant expression in the following #if directive and if statement is not guaranteed to evaluate to the same value in these two contexts: #if ’z’ - ’a’ == 25 if (’z’ - ’a’ == 25) — end note] Also, whether a single-character character literal may have a negative value is implementationdefined. Each subexpression with type bool is subjected to integral promotion before processing continues.

AnInowledgments