Philippe Dunski
dunski.philippe@gmail.com

Disambiguating overflow/underflow constexpr evaluation

Abstract

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

Overflow/ underflow for every integer type different than unsigned int is currently an undefined behaviour, when `UINT_MAX` + 1 and `UINT_MIN - 1` are equal to 0

This proposal aims to fix some cases of this undefined behaviour, making compiler more consistant when dealing with integral `constexpr` evaluation, making such evaluation clearly and explicitly conceptual.

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` assignment's result should be constrainted by its underlying type range, regardless of whether this type is `signed` or `unsigned`, and regardless of the expression's type at the right side of the assignment operator.

The problem

There is, actually, three main line defineing the compiler's behaviour when dealing with integer constexpr value definition :

- 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` (but 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 – two good reasons to consider the runtime behaviour to be bad for `constexpr`.

1- 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 their error list with some specific options
- Some compilers are just fine with such code and will only produce a warning if some specific option is set
- some compiler have different behaviour following the way the value is expressed ; agreeing with the use of `std::numeric_limits<some_type>::max()+1`, but producting a compile time error when using directly the corresponding value

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

The main reason for this situation is that at least 8 of those cases are undefined behaviour.

An « unexpected loop » in constexpr values

Even worst than compiler inconsistancy is the possible presence of an unexpected – and bug prone – loop in `constexpr` evaluation.

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 list which could be defined in the form of

```
template <typename ... Args>

struct type_list{};
```

Everything seems to be perfectly fine, isn't it ? But, please, 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_type, type_index>;
```

The 256th type's index could be computed to 0, since `CHAR_MAX+1` is an undefined behaviour. But their is already an index equal to 0 (it should be the index of the first type in the list).

We get an « unexpected loop » when definiing our indexes value, making every index to possibly represent more than one item.

Such situation is – clearly – inacceptable.

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 this result doesn't overflow from promoted type point of view.

I'm in peace with such argument ... when speaking about **runtime**.

But, here, I'm specifically speaking about `constexpr`. In other words, I'm speaking about **compiler's constant, evaulated at compile time**. Promotion is just ... irrelevant here.

Currently, 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, for any reason, computed value does'nt fit in that range, compiler **should have only one possible answer** : to fail at compile time.

And developer should be advised that he has choosen an integer type which is to small to represent the current result.

Incompatibilities issues

Another could argue that

> Changing this could cause an incompatiblity with C.

As far as we are speaking about compile time constants, whose are defined only at the compiler's level, I don't think there could be any compatibility issue : When a C compiler is always able to use defined value just as all other compile time constant.

breaking code issues

In worst case, we new rule can break some existing code working on some specific compiler due to the undefined behaviour.

Fixing such undefined behaviour can, of course, break such code, making it to fail at compile time, because new defined behaviour is more strict.

I expect that such strictness would mainly fix some hidden or undiscovered bugs (cf : « unexpected loops »)

Proposed wording

To fix this issue, some change or addition should be required in at least five places

- 6.7.4 [conv rank]
- 7,6 [conv.prom]
- 8.6 [expr.const]
- 10.1.5 [dcl.constexpr]
- 19.1.10 [cpp.cond] :

Addition in 6.7.4 [conv.rank]

1.11) *– conversion rank applies on constexpr evaluation only if in the context of the constexpr, we can determine that the result type of the whole constexpr expression is greater than every constexpr type in the expression*

Addition in 7.6 [conv.prom]

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

> Promotion can apply in constexpr evaluation only if result type isat least large enough to represent the promoted value
>
> ex :

```
constexpr auto i = std::numeric_limit<uint8_t>::min()-1; // error due to underflow
constexpr auto some_function(){
    return std ::std::numeric_limits<uint16_t>::max()+1; // error due to overflow
}
constexpr std ::int16_t = CHAR_MAX + 1 ; // ok : promotion from char to int16_t allowed
```

(nota : as far as such rule 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 form of :

> Integer `constexpr` evaluation should be constrainted by the contextual result type's range, without any opportunity to overflow nore underflow, no mater if underlying type is `signed` or `unsigned` .

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

example in 8.6.2.23 :

```
constexpr int y = h(1); // OK: initializes y with the value 2
                        // h(1) is a core constant expression because
                        // the lifetime of k begins inside h(1)
constexpr int16_t = h(65535); // error: core constant (65536) doesn't
                              // fit in int16_t's range
```

Modification to 10.1.5 [dcl.constexpr]

> *1) - The constexpr specifier shall be applirf only to the definition of a variable or variable template or the declaration of a function or function template.* ***Its result type is purely contextual****. A function or static data member declared with the constexpr specifier is implicitely an inline function or variable (10.1.6). If any declaration oof a function or function template has a constexpr specifier, then all its dclarations shall contain the constexpr specifier. [Note : <left unchaged>]*

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 occurs in an integer constepr evaluation, 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.

NOTE : more adjustements could be needed throug the standard, especally regarding the narrowing clauses and rules.

Aknowledgments