WG 14, N2466 (fka N2428)

Towards Integer Safety

David Svoboda
svoboda@cert.org

Date: 2019-12-06

Change Log

2019-12-06:

- s/checked_/ckd_/g; in proposed API
- Added 'Extensions' section which lets us delegate extensions to subsequent proposals
- API now uses naming conventions for integer types derived from atomics (C17, s7.17.6)
- Added integer types for fixed-size integers (eg. uint32_t, etc) to API
- Added normative text
- Clarified overflow & wrapping to match usage in C17
- New document number

The Problem

Because integers have fixed ranges, arithmetic operations on them can cause unexpected wrapping or overflow. Unsigned integers display modular behavior. While this behavior is well-defined, it is often unexpected. Signed integers also frequently display modular behavior, but signed integer overflow is actually undefined behavior. Many real-world vulnerabilities and exploits arise from signed integer overflow or unsigned integer wrapping (CVE-2009-1385 and CVE-2014-4377 among many others).

After studying the current state-of-the-art in integer safety in C and other languages, we decided that this proposal should be low-level; it should provide access to operations that detect overflow. We therefore leave room for subsequent proposals to build on our proposal, perhaps at providing cleaner syntax or more extensive functionality.

Convention

The C17 standard does not define overflow or wrap-around / wrapping. But these terms are used often enough that their specific definitions can be inferred. We strive to follow the C17 conventions when using these terms.

In C17, 'overflow' is a condition where the result of an operation cannot be represented in the associated type of the operation result. Both signed and unsigned integer operations may overflow. Silent wrap-around is a behavior that can occur as a result of overflow.

Confusingly, 3.4.3 p3 states:

> EXAMPLE An example of undefined behavior is the behavior on integer overflow.

However, 6.2.5 p9 clarifies unsigned integer behavior:

> A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

Conventionally, signed integer overflow is considered undefined in C but unsigned integer overflow is defined to silently wrap.

Related Work

There have been several attempts to provide safe integer operations:

GCC Built-Ins

GCC provides a handful of non-standard intrinsic functions for performing safe arithmetic. They are documented at https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html

These functions return a boolean value indicating whether overflow occurred in the computation. They also store the solution in a pointer passed to the function. For example, this function:

bool **__builtin_sadd_overflow** *(int a, int b, int *res)*

operates on signed ints. There are similar functions for longs and long longs, as well as for unsigned types. There is also a **__builtin_add_overflow()** macro that takes three parameters and delegates them to the appropriate function based on their type.

There are also analogous functions for doing safe subtraction and multiplication. However, GCC provides no support for division, modulo, or left or right shift operations.

Because these functions store the result in a pointed-to value, they are not suitable for compile-time arithmetic, and embedding them into expressions (such as multiplying the sum of two numbers with the subtraction of two more) is cumbersome.

These functions cannot be used to produce compile-time constants.

Clang provides the same functions and macros described above as GCC. They are documented at:
https://clang.llvm.org/docs/LanguageExtensions.html#checked-arithmetic-builtins

MS Visual C has similar C functions in their **intsafe.h** header file:
https://docs.microsoft.com/en-us/windows/win32/api/intsafe/

Supplemental GCC Built-Ins

For compile-time operations, GCC provides several additional functions, which are not available in Clang or MS Visual C:

bool **__builtin_add_overflow_p** *(type1 a, type2 b, type3 c)*

This macro operates like the **__builtin_add_overflow()** , but it does not actually compute the solution or store it. It merely returns whether the solution would overflow. It uses the final parameter as the type

that the solution should occupy to determine overflow. As such, it overcomes the compile-time limitations of __builtin_add_overflow().

The SafeInt Library

This is a platform-independent library written by David LeBlanc for providing integer safety: https://archive.codeplex.com/?p=SafeInt

The SafeInt library is implemented in C++using C++ templates. This shortens the code, as these templates can apply to multiple integer types. C++'s operator overloading also allows the safe operations to use the same operators as unsafe operations. That is, a+b is a safe operation if a and b are safe integers.

SafeInt has been bundled with MS Visual Studio:
https://docs.microsoft.com/en-us/cpp/safeint/safeint-library?view=vs-2019

Boost Safe Numerics Library

This is a library for handling safe integers, based on SafeInt:

https://github.com/boostorg/safe_numerics

Having evolved from SafeInt, it shares many of the pros and cons of SafeInt.

Before being integrated into Boost, Robert Ramey proposed adding this library to C++'s standard library:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0228r0.pdf

Java Math Exact Methods

In 2014, Java 8 was released. One of its new features was the set of *exact* calculation methods in the Math class. They either return a mathematically correct value or throw an ArithmeticException if overflow occurs.

These methods provide overflow checking for addition, subtraction, and multiplication, as well as increment and decrement. There are no "exact" methods for division, remainder, or shift operations. There are methods to operate on Java int types and Java long types.

Java's +, -, * operators remain unchanged...they will still silently wrap if the mathematical solution cannot be represented by the expression type. (Java operations mandate two's-complement semantics.)

More information is available at:

https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

Approach

Our approach depends on a number of factors:

Invention

While the committee's charter discourages invention, what constitutes "invention" is unclear. Is it invention to adopt **__builtin_sadd_overflow()**, implemented in GCC & Clang, but rename it? What if we reorder the arguments? What if we make it produce compile-time constants? We feel the function

is not suitable for standardization as is, but we could standardize something with the same functionality but a better signature.

Ease of use

Computing a complicated mathematical equation such as a * b + c * d becomes cumbersome when using functions (or function-like macros) to perform the math. Preserving an "overflow bit" complicates things further. A solution that merely provided functions without overloading operators would be cumbersome. For example: add(multiply(a, b), multiply(c, d)) is harder to read, even without considering the possibility of overflow.

However, restricting ourselves to functions does have several advantages: Operators introduce ambiguities in the syntax, which are traditionally resolved by precedence order and the associative rule. The associative property of addition implies that (a+b)+c == a+(b+c), which means the additions can be done in either order. Technically C does not guarantee this, because signed integer overflow is undefined behavior, but when signed overflow wraps, the associative rule is preserved. However, the associative rule is also not preserved when considering overflow. (UINT_MAX + 1) – 1 and UINT_MAX + (1 – 1) both produce the same result in mathematical integers, and in C signed integers when overflow wraps. However, the first evaluation overflows, but the second doesn't.

Furthermore, a discussion on the WG14 reflector reveals that everyone has their own approach to integer safety. A one-size-fits-all solution is unlikely to satisfy enough committee members to gain traction.

We therefore choose to forego usability and implement a minimal "bare-bones" solution, upon which everyone can propose more user-friendly options.

An alternative proposal would be to standardize access to overflow bits. The x86 family of processors, as well as many others, will have an 'overflow flag' that indicates if signed integer overflow occurred in the last operation. They also have a 'carry flag' to indicate if unsigned integer wrapping occurred. However, these flags, while common, are not universal. The DEC Alpha lacks them completely, but provides other mechanisms for detecting overflow. Therefore, we must standardize some way of detecting when operations overflow, but we cannot standardize access to these flags.

Extensions

Our decision to produce a core proposal and supplemental proposal allows us to forego many extensions, delegating them to subsequent proposals, and we need only ensure that our core proposal makes them possible.

For example, it has been suggested that we provide overflow checking for atomic types. This could be done, but entails many difficulties dealing with concurrency, and would be best handled as a separate proposal. We employ this same strategy to other suggestions, including operator overloading.

There was also a suggestion to extend this proposal to smaller types like short and char. We chose to ignore those types because of integer promotions, which promote shorts and chars to ints (signed or unsigned) before performing operations on them. Supporting shorts and chars would add extra

portability complications, and hence we encourage others to provide support for them, but will not support them here.

Compile-time evaluation

A solution that can be used to compute compile-time constants is preferable to one that cannot be used at compile-time. Only the GCC supplemental functions provide compile-time constants.

Completeness

The GCC builtins, as well as Java, ignore division, remainder, or shifting operations. They consider only addition, subtraction, and multiplication. We will therefore restrict ourselves to these operations.

Namespace Pollution

It has been suggested that the GCC builtins, by defining many functions pollute the namespace, and a suitable standard proposal would suffer the same fate. This problem is being addressed by N2409, and we will not address it separately here.

Approach Conclusion

Given our design decisions, we have decided to provide a core proposal, and a supplemental proposal. The core proposal is low-level, and not necessarily easy to use. But it serves as a suitable foundation to provide friendlier APIs for the same functionality. Other proposals, such as a 'checked' qualifier to address integer types, can leverage the core proposal.

The supplemental proposal does exactly this: it leverages the core proposal. Hence it is worthwhile only if the core proposal is acceptable. It requires no additional intrinsic functions, and could be implemented as a few additional headers and macros.

Core Proposal

The core proposal is based on standardizing the GCC Builtins, with addressing their shortcomings. That is, they will have acceptable names and signatures, and they can produce constant expressions.

We first propose a type to represent checked integers:

```
ckd_$TYPE_t
```

This type provides access to its value, as well as access to an overflow bit. It could be implemented as a struct, but need not be.

Here $TYPE represents the type of integer value, as indicated in the following table:

$TYPE	**Type**
`int`	`signed int`

$TYPE	**Type**
uint	unsigned int
long	signed long
ulong	unsigned long
llong	signed long long
ullong	unsigned long long
intmax	intmax_t
uintmax	uintmax_t
size	size_t
ptrdiff	ptrdiff_t
intptr	intptr_t
uintptr	uintptr_t
intN	intN_t
uintN	uintN_t

The last two types employ a size N, and indicate a signed or unsigned integer of exactly N bits. The precise set of values for N for which signed or unsigned checked integer types are defined is implementation-dependent.

The following function-like macros provide access to the contents of this type. Note that the contents need not be addressable. The macro

```
bool ckd_overflow(x)
```

returns true if x's overflow flag has been set. The macro

```
$TYPE ckd_value(x)
```

returns x's value. If the overflow flag is clear, x's value is implied to correctly represent the mathematical value of whatever operation(s) produced x. If the overflow flag is set and the type is signed, then x's value is unspecified. If the type is unsigned, then x's value is the expected result of modular arithmetic. (If the C committee adopts two's-complement representation, then instead the value will be specified to be the expected two's-complement result.) (On platforms with twos-complement arithmetic, x might represent the lower-order bits of the mathematically correct value.)

This core proposal does not provide any 'constructors' for the checked types... constructors are, however, provided in the supplemental proposal.

We further propose a family of functions that perform operations on integers and returned a checked type:

```
ckd_$TYPE_t ckd_$TYPE_$OP($TYPE a, $TYPE b);
```

Here $TYPE represents the type of integers, and has one of the values defined above.

$OP represents the operation involved, and can be one of the following:

$OP	**Operation**
`add`	+ (Addition)
`sub`	– (Subtraction)
`mul`	* (Multiplication)

In the resulting checked type, overflow indicates whether overflow occurs. Value indicates the operation's resulting value. If overflow is false then value represents the correct mathematical result of the operation. If overflow is true, then value is unspecified. We would recommend that if overflow is true, then value is the same value that would have resulted from applying the operations on unchecked integers.

If both integers are constant expressions, then the returned type is also a constant expression.

For example, to add two signed ints, this function would be used:

```
ckd_int_t ckd_int_add(int a, int b);
```

These functions incur little overhead. On x86 platforms, they only require fetching the overflow or carry flag upon completion of their operation.

Supplemental Proposal

The supplemental proposal builds on top of the functions defined in the core proposal.

The following functions (or function-like macros) can be used to construct a checked value:

```
ckd_$TYPE_t make_ckd_$TYPE_t($TYPE x, bool flag);
```

This explicitly constructs a checked integer type given the plain integer and an overflow flag (which will typically be false, indicating that the value is correct. However, a true overflow flag could be useful to explicitly indicate an overflow error inside an expression).

The proposal also provides these more sophisticated functions:

```
ckd_$TYPE_t ckd_c$TYPE_$OP(ckd_$TYPE a, ckd_$TYPE b);
ckd_$TYPE_t ckd_c$TYPE_$TYPE_$OP(ckd_$TYPE a, $TYPE b);
ckd_$TYPE_t ckd_$TYPE_c$TYPE_$OP($TYPE a, ckd_$TYPE b);
```

Again $OP is one of "add", "sub", or "mul", and $TYPE is an integer type as defined in the core proposal. For example, the function to add a checked signed int to an unchecked signed int would be:

```
ckd_cint_t ckd_cint_int_add(ckd_int_t a,int b);
```

The value in the return type is the mathematical result of the operation, as if it were performed with no checks. The overflow in the return type is true if the operation overflows or either overflow flag in the arguments is true. For functions where either argument is an unchecked integer, the argument is converted to a checked integer type, by setting the value to the argument, and the overflow flag to false.

These functions allow the developer to mix and match checked and unchecked integer types while providing overflow checks. Note that we only provide functions that work on identical integer types; there are no functions that mix integer types, such as signed int and unsigned long.

Finally, we also propose the following macros:

```
ckd_add(x,y)
ckd_sub(x,y)
ckd_mul(x,y)
```

Each macro is generic and delegates work to the appropriate checked function based on the type of x and the type of y. That is ckd_add() calls ckd_int_add() if x and y are ints, ckd_long_clong_add() if x is a long and y is a checked long int, and so on.
While not as elegant as using operators, this does provide some syntactic simplicity in its usage. To check (a + b) * (c + d), one can write:

```
ckd_mul(ckd_add(a,b), ckd_add(c,d))
```

The result will have an overflow bit that correctly indicates if either addition or multiplication overflowed, regardless of the checked-ness of the types of a, b, c, or d. If it is false, the result will be mathematically correct.

Note that these macros assume all arguments are the same integer type, and only allow variation between checked vs. unchecked types. Mixing distinct integer types must be handled externally, perhaps using type casting. This is an intentional feature omission, because type conversion can lead to loss of precision or misinterpretation of sign, which our design does not address.

Proof of Concept

To verify that the supplemental proposal is feasible, we provide the following code. This code uses the __builtin_add_overflow() function from GCC Builtins, and should compile with a sufficiently new version of GCC or Clang. It implements the ckd_add() macro, although it only considers its parameters to be either signed ints or checked signed ints. It also does not address the requirement of the result being a compile-time constant if the arguments are compile-time constants.

```
// Prints (on 64-bit RHEL7.5):
// Sum is: 2147483646, overflow is 1

#include <limits.h>
#include <stdio.h>
#include <stdbool.h>

typedef struct ckd_int_s {
  bool overflow;
  int value;
```

```
} ckd_int_t;

#define make_ckd_int_t(x) ((ckd_int_t) {false, x})

ckd_int_t
ckd_cint_cint_add(ckd_int_t x, ckd_int_t y) {
  ckd_int_t result;
  result.value = 0;
  result.overflow =
    __builtin_add_overflow( x.value, y.value, &(result.value))
    || x.overflow || y.overflow;
  return result;
}

ckd_int_t
ckd_cint_int_add(ckd_int_t x, int y) {
  return ckd_cint_cint_add(x,make_ckd_int_t(y));
}

ckd_int_t
ckd_int_cint_add(int x, ckd_int_t y) {
  return ckd_cint_cint_add(make_ckd_int_t(x),y);
}

ckd_int_t
ckd_int_int_add(int x, int y) {
  return ckd_cint_cint_add(make_ckd_int_t(x),make_ckd_int_t(y));
}


#define ckd_add(x,y)                                          \
  _Generic((x),                                               \
           ckd_int_t: (_Generic((y),                          \
                       ckd_int_t: ckd_cint_cint_add,          \
                       int: ckd_cint_int_add,                 \
                       default: NULL /* error */)),           \
           int:  (_Generic((y),                               \
                       ckd_int_t: ckd_int_cint_add,           \
                       int: ckd_int_int_add,                  \
                       default: NULL /* error */)),           \
           default: NULL /* error */)                         \
  (x,y)


int main() {
  int x = INT_MAX;
  int y = 1;
  int w = -2.0;
  ckd_int_t z = ckd_add( ckd_add( x, y), w);
  printf("Sum is: %d, overflow is %d\n", z.value, z.overflow);
  return 0;
}
```

Proposed Wording Changes

Core Proposal

Add a new section to chapter 7 before "Future Library Directions":

7.31 Checked Integer Arithmetic <ckdmath.h>
7.31.1 Introduction

1 The header <ckdmath.h> defines macros and declares functions that support checked arithmetic operations.

2 The following integer types support checked integer arithmetic. Each type has a key that appears for functions that support the type:

Type	**Key**
`signed int`	`int`
`unsigned int`	`uint`
`signed long`	`long`
`unsigned long`	`ulong`
`signed long long`	`llong`
`unsigned long long`	`ullong`
`intmax_t`	`intmax`
`uintmax_t`	`uintmax`
`size_t`	`size`
`ptrdiff_t`	`ptrdiff`
`intptr_t`	`intptr`
`uintptr_t`	`uintptr`

3 In addition, exact-width integer functions and types may exist for certain widths. The precise set of widths supported by exact-width integer functions is implementation-defined. For each width N, a platform may support any subset of the following types:

Type	**Key**
`int`*`N`*`_t`	`int`*`N`*
`uint`*`N`*`_t`	`uint`*`N`*
`int_leastN_t`	`int_leastN`
`uint_leastN_t`	`uint_leastN`
`int_fastN_t`	`int_fastN`
`uint_fastN_t`	`uint_fastN`

4 For each integer type that supports checked integer arithmetic, the type

`ckd_`*`type`*`_t`

is a complete object type that indicates a checked value. This includes an integer value and a flag that indicates if overflow occurred when computing the value. The "*`type`*" in "`chk_`*`type`*`_t`" is taken from the Key column in the above tables.

5 EXAMPLE The `ckd_ulong_t` type indicates a checked value of type `unsigned long`.

7.31.2 Checked Accessor Macros

7.31.2.1 The `ckd_overflow` Macro

Synopsis
1
```
#include <ckdmath.h>
bool ckd_overflow(ckd_type_t x);
```

Description
2 If `x` is a checked integer, the `ckd_overflow` macro indicates if `x` was computed using an operation that overflowed .

3 If the argument is a constant expression, then the returned object shall also be a constant expression.

Returns
4 The `ckd_overflow` macro returns `true` if overflow occurred when `x` was computed and `false` otherwise.

7.31.2.2 The `ckd_value` Macro

Synopsis
1
```
#include <ckdmath.h>
type ckd_value(x);
```

Description
2 If `x` is a checked integer, the `ckd_value` macro indicates the value of `x`.

3 If the overflow flag is clear, the value correctly represents the mathematical value of whatever operation(s) produced `x`. Otherwise, the value of `x` is the expected result of modular arithmetic on two's-complement representation with silent wraparound on overflow.

4 If the argument is a constant expression, then the returned object shall also be a constant expression.

Returns

5 The `ckd_value` macro returns the value of `x`.

7.31.3 Checked Arithmetic Functions

Synopsis
1
```
#include <ckdmath.h>
ckd_type_t ckd_type_add(type a, type b);
ckd_type_t ckd_type_sub(type a, type b);
ckd_type_t ckd_type_mul(type a, type b);
```

Description
2 These functions perform an arithmetic operation (addition, subtraction, or multiplication) on its arguments, and stores the result in the return value. The return type also indicates if overflow occurs.

3 If the overflow flag of the returned object is clear, the value correctly represents the mathematical value of the operation. Otherwise, the value of the returned object is the expected result of modular arithmetic on two's-complement representation with silent wrap-around on overflow.

4 If both arguments are constant expressions, then the returned object shall also be a constant expression.

5 EXAMPLE To add two values of type unsigned long, this function would be used:

```
ckd_uint_t ckd_ulong_add(unsigned long a, unsigned long b);
```

If either argument is not already an unsigned long, they will undergo integer promotion and the usual arithmetic conversions before being passed to this function.

Returns
6 These functions return a checked type that indicates the result of computation as well as an overflow indicator.

Add the following to chapter 7 in the "Future Library Directions" section:

7.32.18 Checked Arithmetic Functions `<ckdmath.h>`

1 Type and function names that begin with `ckd_` may be added to the declarations in the <ckdmath.h> header.

Finally, Section 6.6 p3 currently says:

3 Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated.

This should be changed to:

3 Constant expressions shall not contain assignment, increment, decrement, or comma operators, except when they are contained within a subexpression that is not evaluated. Constant expressions shall also not contain function calls, unless this Standard expressly permits the particular function to occur in a constant expression, or the expression is contained within a subexpression that is not evaluated.

Supplemental Proposal

In addition to the instructions in the Core Proposal, add the following to section 7.31, before "Checked Accessor Functions":

7.31.2 Checked Value Creation Functions

Synopsis
1
```
#include <ckdmath.h>
ckd_type_t make_ckd_type_t(type x, bool overflow);
```

Description
2 These functions explicitly construct a checked integer type given the unchecked integer and an overflow flag.

3 if the overflow flag is `true`, the value is assumed to have involved overflow.* Otherwise the value is assumed to be mathematically correct.

4 If both arguments are constant expressions, then the returned object shall also be a constant expression.

Returns
5 These functions return a checked type that presents the value indicated by `x` and the overflow state indicated by `overflow`.

The footnote on 7.31.2p3 should state:

* Constructing a checked integer with an `overflow` flag set to `true` can be useful when explicitly indicating an overflow error inside an expression.

Section 7.31.4 (which was 7.31.3 in the core proposal) should be replaced with the following sections:

7.31.4 Checked Arithmetic Functions

Synopsis
1
```
#include <ckdmath.h>
ckd_type_t ckd_type_add(type a, type b);
ckd_type_t ckd_type_ctype_add(type a, ckd_type_t b);
```

```
ckd_type_t ckd_ctype_type_add(ckd_type_t a, type b);
ckd_type_t ckd_ctype_add(ckd_type_t a, ckd_type_t b);

ckd_type_t ckd_type_sub(type a, type b);
ckd_type_t ckd_type_ctype_sub(type a, ckd_type_t b);
ckd_type_t ckd_ctype_type_sub(ckd_type_t a, type b);
ckd_type_t ckd_ctype_sub(ckd_type_t a, ckd_type_t b);

ckd_type_t ckd_type_mul(type a, type b);
ckd_type_t ckd_type_ctype_mul(type a, ckd_type_t b);
ckd_type_t ckd_ctype_type_mul(ckd_type_t a, type b);
ckd_type_t ckd_ctype_mul(ckd_type_t a, ckd_type_t b);
```

Description
2 These functions perform an arithmetic operation (addition, subtraction, or multiplication) on its arguments, and stores the result in the return value.

3 The return type also has an overflow flag. The overflow flag is set if overflow occurred in the operation or either argument was a checked type whose overflow flag was set.

4 If the overflow flag of the returned object is clear, the value correctly represents the mathematical value of the operation. Otherwise, the value of the returned object is the expected result of modular arithmetic on two's-complement representation with silent wrap-around on overflow.

5 If both arguments are constant expressions, then the returned object shall also be a constant expression.

6 EXAMPLE To add two values of type signed int, this function would be used:

```
ckd_int_t ckd_int_add(int a, int b);
```

7 EXAMPLE To multiply two values of type ckd_ulong_t, this function would be used:

```
ckd_ulong_t ckd_culong_culong_mul(ckd_ulong_t a, ckd_ulong_t b);
```

Returns
8 These functions return a checked type that indicates the result of computation as well as an overflow indicator.

7.31.5 Generic Checked Operation Functions

Synopsis
1
```
#include <ckdmath.h>
ckd_type_t ckd_add(a, b);
ckd_type_t ckd_sub(a, b);
ckd_type_t ckd_mul(a, b);
```

Description
2 These generic functions perform addition, subtraction, or multiplication by invoking a suitable checked operation function, based on the type of `a` and the type of `b`.

3 Both `a` and `b` must be integer types of the same width and signedness. They may be checked or unchecked.

4 The return type also indicates if overflow occurred in the operation or either argument was a checked type whose overflow flag was set.

5 If the overflow flag of the returned object is clear, the value correctly represents the mathematical value of the operation. Otherwise, the value of the returned object is the expected result of modular arithmetic on two's-complement representation with silent wrap-around on overflow.

6 If both arguments are constant expressions, then the returned object shall also be a constant expression.

7 EXAMPLE If `a` and `b` are values of type signed int, then

```
chk_sub(a, b);
```

returns a `ckd_int_t` that indicates their difference, and whether computing the difference resulted in overflow. It produces the same result if either `a`, `b` or both are checked ints with unset overflow flags.

Returns
8 These macros return a checked type that indicates the result of computation as well as an overflow indicator.

Acknowledgements

This proposal was suggested by Dr. Will Klieber.

Special thanks to Martin Sebor, Aaron Ballman, Jens Gustedt, Robert Seacord, and Will Klieber for reviewing this document and making suggestions.