WG 14, N2543 (fka N2466)

Towards Integer Safety

David Svoboda
svoboda@cert.org

Date: 2020-07-08

Change Log

2020-06-01:

- Incorporated heterogeneous capability (to add differently-type integers), which more closely resembles GCC/Clang behavior. Still a core + supplemental proposal but their contents have been reorganized.
- Checked integer arithmetic now apply to shorts and signed & unsigned characters (but not plain char).
- Proposed functions, types, and macros now live in <stdlib.h> (because GCC/Clang current solutions live there)
- Eliminated compile-time constant expressions from normative text.
- New section that promotes use of checked/ckd_ over other terminology choices
- New section that addresses compatibility with N2501 (extended integer types)
- Updated proof-of-concept implementation, added discussion about extending it, including N2501 compatibility.
- Removed most type-specific functions, leaving just the macros. (They could rely on type-specific functions, but that is a quality-of-implementation issue.) There are now a handful of type-specific functions for the most common types, which reflects GCC's implementation.

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 (e.g. 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 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.

This has been addressed by clarity request N2517.

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. These types need not be identical. If they differ, then the result is true if the result cannot be expressed in the result's type, which could be due to overflow, a truncation error or a misinterpretation of sign. That is, the types may be heterogeneous.

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.

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:

Terminology

Any types, functions, and macros that we propose should use a term to distinguish operations that detect overflow from operations that do not. The following candidates are plausible terms:

Term	Source	Context
Checked	Clang	Informally called "checked arithmetic builtins"
Overflow	GCC / Clang	All builtins end with "_overflow"
Safe	LeBlanc / Ramey	Proposed "safe<>" template for integer types
Exact	Java 8	Operation methods end in "Exact"

Of these terms, the term "overflow" is the most precise. As noted earlier, overflow is, in C parlance, something that can happen with both signed and unsigned integers, and our types and operations prevent overflow and nothing else. Nonetheless, referring to them as "non-overflowing" types and methods is awkward. Furthermore, overflow is not the only possibility…values could be truncated during conversion, and misinterpretation of sign can occur, meaning that "overflow" is no longer correct. However, referring to operations as checked types and operations (and the others as unchecked types and operations) is straightforward and intuitive, albeit less precise. We will therefore use "ckd_" as a prefix when designating those types and operations that detect overflow, truncation, or misinterpretation of sign. Any flag that determines if a number was computed correctly will be called an 'exact' flag.

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 constant expressions? 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.

Compatibility with N2501

N2501 proposes adorning ISO C with "extended" integers, that support an arbitrary width. They behave much like bit-fields, and they can have a minimum of 1 value bit, with no official maximum, although unofficially they may support millions of value bits. While these integers support all of the standard C operators, operations on extended integers are currently unchecked. A proposal to endow extended integers with checked types and operations would be useful should both this proposal and N2501 be standardized. Such a proposal would be useful, but should be distinct from both our core and supplemental proposals.

We can address some technical details in the "Proof of Concept" section.

Compile-time evaluation

A solution that can be used to compute compile-time constant expressions is preferable to one that cannot be used at compile-time. Only the GCC supplemental functions provide compile-time constant expressions. However, this can also be a quality-of-implementation issue; many platforms may choose to make their checked arithmetic provide constant expressions, and we should not prevent them from doing so.

We have decided to forgo standardizing compile-time constant expressions in our checked arithmetic, and leave it as a quality-of-implementation issue.

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.

Functions vs. Macros

There has been some debate over whether we should use functions or macros to provide checked integer arithmetic. Using functions confers the following advantages:

- A compiler can verify the correctness of function argument types.
- Function return values can be non-ignorable, thanks to the new [[nodiscard]] attribute
- Functions can be invoked from other languages that provide a bridge to C function calls.

Using macros confers the following advantages:

- Since macros are type-independent, one macro can replace a family of functions, by servicing many different types of arguments.

The builtins used by GCC and Clang provide both sets of advantages, because they are not implemented as functions or macros. However, we cannot prescribe that the compile magic they utilize can or should be ported to every C platform.

We have therefore provided a compromise: We use macros in most places to achieve a simpler API and type independence. We have also provided a few functions for the most common cases where people might want access from other languages.

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.

This proposal consists of the following macros:

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

Each macro performs its operation on two unchecked integers x and y, fills result with the result of the computation, and returns true if the computation is valid. Both x and y may be any integer type, and result is a pointer to an integer of any type. If it were a function, the signature of ckd_add() would be:

```
bool ckd_add(int_type1 *result, int_type2 x, int_type3 y);
```

A platform might implement these macros using functions (eg __ckd_int_add(), __ckd_long_add(), etc), but is not required to.

The result will be the result of the computation. For ckd_add(), result will have the same value as x + y, if defined.

The return value will be true unless the operation produces a result that cannot be represented in the type implied by usual arithmetic conversions, or converting that result to the type indicated by the first argument results in truncation or misinterpretation of sign. If the return value is true, then result actually points to the correct mathematical value of the operation. If the return value is false, then the value that result points to is indeterminate; it might be the low-order bits of the correct mathematical value, or the result of sign misinterpretation, depending on the behavior of unchecked overflow.

These macros have several properties:

- They are already supported with minimal changes in GCC and Clang.
- They do not require a ‘checked integer type’
- They check for overflow as well as any conversion error when storing the result
- They cannot be chained together. An expression like a + b * c requires two non-overlapping macro calls.

In addition to these macros, we also provide a handful of functions to do the same thing. Each function takes operands of a single type, and computes a result using that type. These functions are [[nodiscard]], which provides some security that their return value must not be ignored. Finally, as functions, they can be called from other languages.

To mimic the GCC implementation, we only provide functions to operate on signed and unsigned ints, longs, and long longs.

Supplemental Proposal

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

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 ‘exact’ flag. 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`
`uint`	`unsigned int`
`long`	`signed long`
`ulong`	`unsigned long`
`llong`	`signed long long`
`ullong`	`unsigned long long`
`char`	`signed char`
`uchar`	`unsigned char`
`short`	`signed short`
`ushort`	`unsigned short`
`intmax`	`intmax_t`
`uintmax`	`uintmax_t`
`size`	`size_t`

$TYPE	**Type**
ptrdiff	ptrdiff_t
intptr	intptr_t
uintptr	uintptr_t
intN	intN_t
uintN	uintN_t

Note that plain char is not supported…only signed and unsigned chars may undergo checked arithmetic operations.

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_exact(x)
```

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

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

returns x's value. If the exact flag is set, x's value is implied to correctly represent the mathematical value of whatever operation(s) produced x. If the exact flag is clear 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 the value will be specified to be the expected two's-complement result regardless of the type's signedness.) (On platforms with twos-complement arithmetic, x might represent the lower-order bits of the mathematically correct value.)

The following functions can be used to construct a checked value:

```
ckd_$TYPE_t make_ckd_$TYPE_t($TYPE value, bool exact);
```

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

The following macros from the core proposal:

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

are enhanced. In the supplemental proposal, x and y can be any integer type or any checked integer type. Likewise, result may be a pointer to any integer type or checked integer type. If result points to a checked integer type, the exact flag of result is set to the same Boolean value that is returned.

To complete the supplemental proposal, these macros can also take the following forms:

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

Each macro performs its operation on two integers, checked or unchecked, and returns a result as a checked integer. Both x and y may be any integer type or any checked integer type. The resulting type of the checked integer's value is based on the usual arithmetic conversions, as described in C17 s6.3.1.8.

The two-argument macro forms have several properties:

- They do less than the three-argument macro forms. They check for overflow, but do not indicate conversion errors.
- They do require the ckd_*$TYPE*_t type to be defined.
- These macros allow chaining. That is, the expression ckd_add(a, ckd_mul(b, c)) will compute a + b * c, and indicate if any error occurs.
- The type of the expression follows the usual arithmetic conversions.

Since checked integer types do not provide any type conversions, the result of these macros cannot be assigned to a plain integer type, or a checked integer of the wrong type. Thus, these macros provide type-safety.

Proof of Concept (Type-Generic Macros)

To verify that the supplemental proposal is feasible, we provide the following code. This code uses the __builtin_add_overflow() function from GCC, and should compile with a sufficiently modern version of GCC or Clang. It implements the ckd_add() macro from the supplemental proposal, although it only considers its parameters to be 32-bit signed ints, checked or unchecked.

```
// Prints (on 64-bit RHEL7.5 and MacOS 15.4):
// Sum is: 2147483646, overflow is 1

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

/* T is an exact-width integer type, which __builtin_add_overflow
   supports */

#define CKD(T) ckd_ ## T
#define DEFINE_CKD_TYPE(T)                      \
  typedef struct ckd_s_ ## T {                  \
    bool overflow;                              \
    T value;                                    \
```

```
  } CKD(T)

#define make_ckd(T, x) ((ckd_ ## T) {false, x})

#define CKD_ADD_CKD_CKD(T) ckd_add_ckd_ ## T ## _ckd_ ## T
#define DEFINE_CKD_ADD_CKD_CKD(T)                                       \
  CKD(T)                                                                \
    CKD_ADD_CKD_CKD(T)(CKD(T) x, CKD(T) y) {                            \
    CKD(T) result;                                                      \
    result.value = 0;                                                   \
    result.overflow =                                                   \
      __builtin_add_overflow( x.value, y.value, &(result.value))       \
      || x.overflow || y.overflow;                                      \
    return result;                                                      \
  }

#define CKD_ADD_CKD_UNCKD(T) ckd_add_ckd_ ## T ## _ ## T
#define DEFINE_CKD_ADD_CKD_UNCKD(T)                                     \
  CKD(T)                                                                \
    CKD_ADD_CKD_UNCKD(T)(CKD(T) x, T y) {                               \
    return CKD_ADD_CKD_CKD(T)(x,make_ckd(T,y));                         \
}

#define CKD_ADD_UNCKD_CKD(T) ckd_add_ ## T ## _ckd_ ## T
#define DEFINE_CKD_ADD_UNCKD_CKD(T)                                      \
  CKD(T)                                                                 \
    CKD_ADD_UNCKD_CKD(T)(T x, CKD(T) y) {                                \
    return CKD_ADD_CKD_CKD(T)(make_ckd(T,x),y);                          \
  }

#define CKD_ADD_UNCKD_UNCKD(T) ckd_add_ ## T ## _ ## T
#define DEFINE_CKD_ADD_UNCKD_UNCKD(T)                                          \
  CKD(T)                                                                       \
    CKD_ADD_UNCKD_UNCKD(T)(T x, T y) {                                         \
    return CKD_ADD_CKD_CKD(T)(make_ckd(T,x),make_ckd(T,y));                    \
  }

#define DEFINE_CKD(T)                           \
  DEFINE_CKD_TYPE(T);                           \
  DEFINE_CKD_ADD_CKD_CKD(T);                    \
  DEFINE_CKD_ADD_CKD_UNCKD(T);                  \
  DEFINE_CKD_ADD_UNCKD_CKD(T);                  \
  DEFINE_CKD_ADD_UNCKD_UNCKD(T);

DEFINE_CKD(int32_t);
DEFINE_CKD(uint32_t);
DEFINE_CKD(int64_t);
DEFINE_CKD(uint64_t);

#define ckd_add(x,y)                                                    \
  _Generic((x),                                                         \
           CKD(int32_t):                                                \
           (_Generic((y),                                               \
                     CKD(int32_t): ckd_add_ckd_int32_t_ckd_int32_t,     \
                     int32_t: ckd_add_ckd_int32_t_int32_t,              \
                     /* ...Address other integer types... */            \
                     default: NULL /* error */)),                       \
           int32_t:                                                     \
           (_Generic((y),                                               \
```

```
                    CKD(int32_t): ckd_add_int32_t_ckd_int32_t,          \
                    int32_t: ckd_add_int32_t_int32_t,                   \
                    /* ...Address other integer types... */             \
                    default: NULL /* error */)),                        \
            /* ...Address other integer types... */                     \
            default: NULL /* error */)                                  \
  (x,y)


int32_t main() {
  int32_t x = INT_MAX;
  int32_t y = 1;
  int w = -2.0;
  CKD(int32_t) z = ckd_add( ckd_add( x, y), w);
  printf("Sum is: %d, overflow is %d\n", z.value, z.overflow);
  return 0;
}
```

This implementation illustrates how checked integers could be implemented without using "compiler magic". It would need to be extended to 64-bit integers and 32-bit unsigned integers. Platform vendors would also want to add a support layer that translates standard types (int, long, etc.) to the exact types and back. Eventually other types (e.g. int128_t) would also need to be supported.

This implementation could also be extended to handle extra types if they are known when it is implemented. The _Generic operator used in the ckd_add() macro must have all types enumerated that it can handle. Adding new types that match one of the listed types requires no more work than mapping the new type to the listed types independently. However, adding a differently-sized type, such as one of the extended integer types from N2501 would require enhancing the _Generic operators in the code, and this would not scale with the potentially millions of new types from N2501. Ideally, some additional power, such as identifying the width of an extended int type would permit a more powerful implementation.

Proof of Concept (2- vs 3-arguement Macros)

To verify that the supplemental proposal is feasible, we provide the following code. This code illustrates how a macro can be overloaded to use variadic forms, complete with type-safety. That is, if add() is invoked with invalid arguments, a compiler error is produced.

```
// Prints (on MacOS)
// Test A: 42
// Test B: 2020

int printf(const char *format, ...);

void add3(int* ret, int x, int y) {
  *ret = x + y;
}

int add2(int x, int y) {
  return x + y;
}

#define add(w, ...)                                 \
  _Generic((w),                                     \
```

```
            int*: add3,                             \
            int: add2                               \
            )(w, __VA_ARGS__)

int main() {
  printf("Test A: %i\n", add(40, 2));
  int temp;
  add(&temp, 2000, 20);
  printf("Test B: %i\n", temp);
}
```

Proposed Wording Changes

Core Proposal

Add a new subsection to section 7.22.6:

7.22.6.3 Checked Integer Arithmetic

7.22.6.3.1 Checked Integer Arithmetic Macros

Synopsis
1
```
#include <stdlib.h>
bool ckd_add(type1 *result, type2 a, type3 b);
bool ckd_sub(type1 *result, type2 a, type3 b);
bool ckd_mul(type1 *result, type2 a, type3 b);
```

Description
2 These generic macros perform addition, subtraction, or multiplication on `a` and `b`, storing the result of the operation in the value pointed to by `result`. In other words, `*result` is assigned the result of computing `a + b`, `a - b`, or `a * b`.

3 Both `a` and `b` must be signed or unsigned character or integer types. They may be any integer type and they need not be the same type. The `result` pointer must reference a value of a signed or unsigned character or integer type.

Returns
4 These macros return true if the type of the first argument is sufficient to hold the result of the computation. If these macros return true, the value assigned to `*result` correctly represents the mathematical result of the operation. If these macros return false, then the type of `*result` is not sufficient to contain the mathematical result. In this case, `*result` is the expected result of modular arithmetic on two's-complement representation with silent wrap-around on overflow.

5 EXAMPLE If `a` and `b` are values of type signed int, and `*result` is a signed long, then
```
   chk_sub(result, a, b);
```
will indicate if `a - b` can be expressed as a signed long. If signed long has a greater width than signed int, this will always be possible and this macro will return true.

7.22.6.3.2 Checked Integer Arithmetic Functions

Synopsis

1

```
#include <stdlib.h>
[[nodiscard]] bool ckd_add_int(int *result, int a, int b);
[[nodiscard]] bool ckd_sub_int(int *result, int a, int b);
[[nodiscard]] bool ckd_mul_int(int *result, int a, int b);

[[nodiscard]] bool ckd_add_long(long *result, long a, long b);
[[nodiscard]] bool ckd_sub_long(long *result, long a, long b);
[[nodiscard]] bool ckd_mul_long(long *result, long a, long b);

[[nodiscard]] bool ckd_add_llong(long long *result, long long a, long
long b);
[[nodiscard]] bool ckd_sub_llong(long long *result, long long a, long
long b);
[[nodiscard]] bool ckd_mul_llong(long long *result, long long a, long
long b);

[[nodiscard]] bool ckd_add_uint(unsigned int *result, unsigned int a,
unsigned int b);
[[nodiscard]] bool ckd_sub_uint(unsigned int *result, unsigned int a,
unsigned int b);
[[nodiscard]] bool ckd_mul_uint(unsigned int *result, unsigned int a,
unsigned int b);

[[nodiscard]] bool ckd_add_ulong(unsigned long *result, unsigned long
a, unsigned long b);
[[nodiscard]] bool ckd_sub_ulong(unsigned long *result, unsigned long
a, unsigned long b);
[[nodiscard]] bool ckd_mul_ulong(unsigned long *result, unsigned long
a, unsigned long b);

[[nodiscard]] bool ckd_add_ullong(unsigned long long *result,
unsigned long long a, unsigned long long b);
[[nodiscard]] bool ckd_sub_ullong(unsigned long long *result,
unsigned long long a, unsigned long long b);
[[nodiscard]] bool ckd_mul_ullong(unsigned long long *result,
unsigned long long a, unsigned long long b);
```

Description

2 These functions perform addition, subtraction, or multiplication on `a` and `b`, storing the result of the operation in the value pointed to by `result`. In other words, `*result` is assigned the result of computing `a + b`, `a - b`, or `a * b`.

Returns

3 These functions return true if the type of the first argument is sufficient to hold the result of the computation. If these functions return true, the value assigned to `*result` correctly represents the mathematical result of the operation. If these functions return false, then the type of `*result` is not sufficient to contain the mathematical result. In this case, `*result` is the expected result of modular arithmetic on two's-complement representation with silent wrap-around on overflow.

Supplemental Proposal

Instead of the instructions in the Core Proposal, add the following new subsection to section 7.22.6. This implies that all checked types, functions, and macros will be available in sdlib.h:

7.22.6.3 Checked Integer Arithmetic

Synopsis
1 The following integer types support checked integer arithmetic. Each direct type has a key that appears for functions that support that type*:

The footnote on 7.22.6.3p1 should state:

* Note that the `char` type does not support checked integer arithmetic.

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

2 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:

Direct 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`

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

`ckd_`*`type`*`_t`

is a complete object type, other than an array type, that indicates a checked value. This includes an integer value and a flag that indicates exactness; that is whether overflow, truncation, or misinterpretation of sign occur when computing the integer value. The "*`type`*" in "`chk_`*`type`*`_t`" is taken from the Key column that corresponds to the direct type in the above tables.*

The footnote on 7.22.6.3p3 should state:

* For example, the `ckd_ulong_t` type indicates a checked value of type `unsigned long`.

7.22.6.3.1 The `ckd_exact` Macro

Synopsis
1
```
#include <stdlib.h>
bool ckd_exact(ckd_type_t x);
```

Description
2 If `x` is a checked integer, the `ckd_exact` macro indicates if `x` was computed using one or more operations that did not produce the mathematically correct result.

Returns
3 The `ckd_exact` macro returns `false` if overflow, truncation, or misinterpretation of sign occurred when `x` was computed and `true` otherwise.

7.22.6.3.2 The `ckd_value` Macro

Synopsis

1
```
#include <stdlib.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 exact flag is set, 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.

Returns
4 The `ckd_value` macro returns the value of `x`.

7.22.6.3.3 The make_ckd_*type*_t functions

Synopsis
1
```
#include <stdlib.h>
ckd_type_t make_ckd_type_t(type value, bool exact);
```

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

3 if the exact flag is `false`, the value is assumed to have involved overflow, truncation, or misinterpretation of sign.* Otherwise the value is assumed to be mathematically correct.

> *The footnote on 7.31.2p3 should state:*
>
> * Constructing a checked integer with an `exact` flag set to `false` can be useful when explicitly indicating an error inside an expression.

Returns
5 These functions return a checked type that represents the value indicated by `value` and the exact state indicated by `exact`.

7.22.6.3.4 Checked Integer Arithmetic Macros

Synopsis
1
```
#include <stdlib.h>
bool ckd_add(type1 *result, type2 a, type3 b);
bool ckd_sub(type1 *result, type2 a, type3 b);
bool ckd_mul(type1 *result, type2 a, type3 b);

ckd_type_t ckd_add(type1 a, type2 b);
```

```
ckd_type_t ckd_sub(type1 a, type2 b);
ckd_type_t ckd_mul(type1 a, type2 b);
```

Description

2 These generic macros perform addition, subtraction, or multiplication on `a` and `b`. In the first form, they store the result of the operation in the value pointed to by `result`, and in the second form, they return the result as a checked integer.

3 Both `a` and `b` must be signed or unsigned character or integer types or checked integer types. They need not be the same type.

4 In the first form, the result pointer must reference a value of a signed or unsigned character or integer type. In the second form, the result will be a checked integer whose type is determined by the usual arithmetic conversions. (Section 6.3.1.8)

5 In the first form, the return type indicates if an error occurred in the operation or either argument was a checked type whose exact flag indicated an error. In the second form, this information is indicated by the exact flag in the return value. If this flag is set, the computed 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.

Returns

5 The first form macros return true if the type of the first argument is sufficient to hold the result of the computation. If these macros return true, the value assigned to `*result` correctly represents the mathematical result of the operation. If these macros return false, then the type of `*result` is not sufficient to contain the mathematical result. In this case, `*result` is the expected result of modular arithmetic on two's-complement representation with silent wrap-around on overflow. If `*result` references a checked integer type then its exact flag will equal the macro's return value.

6 The second form macros return a checked integer type that indicates the result of the computation as well as an exact flag.

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

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

will indicate if `a - b` can be expressed as a signed long. If signed long has a greater width than signed int, this will always be possible and this macro will return true. This behavior occurs whether `result`, `a`, and `b` are checked or unchecked.

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

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

returns a `ckd_long_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 integers with set exact flags.

7.22.6.3.5 Checked Integer Arithmetic Macros

This section is identical to section "7.22.6.3.2 Checked Integer Arithmetic Functions" in the core proposal.

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

7.32.18 Checked Arithmetic Functions `<stdlib.h>`

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

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.

This material is based upon work funded and supported by the U.S. Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

DM20-0381