WG 14, N2683 (fka N2669)

Towards Integer Safety

David Svoboda
svoboda@cert.org

Date: 2021-03-10

Change Log

2021-03-10:

- Many syntactic fixes to the normative text, such as: s/must/shall/g; s/In other words,/, (that is,/g;
- Distinguished between types and variables in normative text
- Added ‘Recommended practice’ sessions to encourage compiler diagnostics for invalid input types.

2021-02-12:

- Consistently used “integer type other than plain `char`, `bool`, or an enumeration type” to indicate the unchecked types supported by the proposed macros.
- The result pointer in the checked integer operations must now reference a “modifiable” lvalue.
- Added to the supplemental proposal that two checked integer types are compatible if their analogous unchecked types are compatible.

2021-01-06:

- There is a new “Usual Arithmetic Conversions” section that addresses some differences in behavior between the C arithmetic operators and the GCC Built-ins. Our checked integers follow the same behavior as the GCC Built-ins, and this is described in the normative text.
- All new definitions now live in a new header: `stdckdint.h`, rather than the familiar `stdlib.h`. They also live in a new section of Chapter 7, instead of a subsection in the “General Utilities” section.
- The checked integer types now have an "inexact" flag rather than an "exact" flag. The inexact flag is true if and only if overflow, truncation, or misinterpretation of sign occurred, and false otherwise. Likewise, the macros that return a flag return true if and only if overflow, truncation, or misinterpretation of sign occurred, and false otherwise. This is in accordance with the GCC built-ins.
- Changed the `make_ckd_`*`type`*`_t()` function names to `ckd_mk_`*`type`*`_t`(). The functions are also marked `[[nodiscard]]`
- Added a `ckd_mk()` macro, which provides the same functionality as the `ckd_mk_`*`type`*`_t`() functions.
- The checked integer types are now “complete object types”.
- Removed “generic” from normative text, in lieu of addressing families of generic functions and macros in separate proposal
- Replaced "signed or unsigned character or integer" with "signed or unsigned integer".
- Added a paragraph to “Approach Conclusion” explaining that we don’t currently allow implicit conversion or typecasting between checked & unchecked integers.
- Incorporated definition of ‘overflow’ and usage of wraparound / wrapping from ISO IEC 10967-1.
- Referenced the results of several straw polls to support paper’s approach.

2020-05-05:

- 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.
- 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.)

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. ISO IEC 10967-1 does define overflow, and uses wrap-around and wrapping, and ISO C complies with these definitions for both integers and floating-point types. Therefore, we follow the ISO IEC 10967-1 and 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.

C17 s6.2.5 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 built-ins”
Overflow	GCC / Clang	All built-ins 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 incorrectly will be called an “inexact” flag.

WG14 conducted a straw poll in the October 2020 virtual meeting. The poll asked if the prefix should be changed from "ckd_" to "checked_". The poll results were 8 in favor, 6 against, and 9 abstaining, which was interpreted as "divided".

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.

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.

Compatibility with N2590 (Adding Fundamental Type for N-bit Integers)

N2590 (fka 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 built-ins, 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 built-ins, 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.

Header File

One minor question is: Which header file, new or pre-existing, should any new functions, macros, or types live in? When first submitted, this proposal suggested a new header called `ckdmath.h`. In the April 2020 virtual meeting, this was changed to the familiar `stdlib.h` due to a shrinkage of new definitions. However, in the October 2020 virtual meeting, people cited that this proposal was complete and well-bounded by itself. There is a clear distinction of what lives in this proposal (and any subsequent proposals) and what does not, therefore this proposal warrants its own header file. After some discussion, we settled on a new header called `stdckdint.h`.

Standard Arithmetic Conversions

The arithmetic operations in standard C can surprise programmers occasionally. Consider this code:

```
signed int32_t MAX_INT = 0x7fffffff;       // max 32-bit value
signed int32_t a1 = MAX_INT + 1;           // Wraparound: a1 = - 2^32
signed int64_t a2 = MAX_INT + 1;           // Wraparound: a2 = - 2^32
signed int64_t a3 = (int64_t) MAX_INT + 1; // No wraparound: a3 = + 2^32
```

This happens because C performs binary integer operations in a type sufficient to represent both operand types, even if this type is not sufficient to hold the mathematical result.

The GCC built-ins circumvent this problem by not using arithmetic conversions. Instead, they use an "infinite precision signed type" to bypass overflow during the operation:

```
signed int64_t a4;
bool flag = __builtin_sadd_overflow( MAX_INT, 1, &a4);
// No wraparound: a4 = + 2^32, flag = false
```

The GCC built-in wording for its operations is:

These built-in functions promote the first two operands into infinite precision signed type and perform addition on those promoted operands. The result is then cast to the type the third pointer argument points to and stored there. If the stored result is equal to the infinite precision result, the built-in functions return false, otherwise they return true. As the addition is performed in infinite signed precision, these built-in functions have fully defined behavior for all argument values.

For compatibility with the GCC built-ins, our checked-integer operations will employ the same strategy. An operation where the result type is specified (implicitly by an argument) will not be hindered by the limits of a type implied by usual arithmetic conversions. However, an operation where the result type is implied (and not specified) will have the type determined by usual arithmetic conversions. So:

```
signed int64_t a5;
bool flag2 = ckd_add( &a5, MAX_INT, 1);  // from core proposal
// No wraparound: a5 = + 2^32, flag = false

ckd_int64_t a6 = ckd_add( MAX_INT, 1); // from supplemental proposal
// Error: ckd_add returns a ckd_int32_t,
// which does not implicitly convert to ckd_int64_t

ckd_int32_t a7 = ckd_add( MAX_INT, 1); // from supplemental proposal
// Wraps around: ckd_value(a7) = -2^32, ckd_inexact(a7) = true
```

There is some question of what terminology to use when adopting this strategy: C17 section 6.5, paragraph 8, footnote 96 uses "infinite range and precision" with regard to intermediate floating-point operations. Many other places in C17also use the terms "infinite range" or "infinite precision", but they all refer to floating-point operations. We can find no usage of these terms when describing integer arithmetic. Nonetheless, we will boldly suggest that our checked integer operations will operate as if they cast their operands to an infinitely-ranged integer type before the operation. Afterwards the result is cast to a suitable resulting type, and this final cast can cause overflow, truncation, or misinterpretation of sign.

One final note: While our approach will use what the GCC built-ins call "infinite-precision" arithmetic, it can only apply such precision to the arguments of our operation functions and macros. An operation like `ckd_add(&result, x+y, z)` will still employ standard arithmetic conversions when adding `x` to `y`, although such conversions will not be used with adding `x+y` to `z`.

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.

Neither the core nor the supplemental proposal provide any implicit conversion mechanisms. You cannot implicitly or explicitly typecast an unchecked integer to a checked integer, and you cannot typecast a checked integer to a checked integer of a different type or signedness. Such conversions could be added in a later proposal, if desired.

Core Proposal

The core proposal is based on standardizing the GCC Built-ins, 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 the value pointed to by result with the result of the computation, and returns false 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 arguments will undergo default integer promotions before being passed to the macro, but will not undergo any implicit conversions, including the usual arithmetic conversions.

The return value will be false 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 false, then result actually points to the correct mathematical value of the operation.

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 the April 2020 meeting, WG14 conducted a straw poll questioning if the committee would prefer a family of addressable functions (instead of macros) to detect integer overflow. The poll results were 5 in favor, 1 against, and 7 abstaining, which constituted no consensus. Hence, we use macros rather than type-specific functions.

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 'inexact' flag. It could be implemented as a struct, but need not be. It is declared as a "complete object type", which allows users to copy one checked int to another of the same type, using assignment or `memcpy()`. Many complete object types, such as `FILE`, provide their own initialization routines, but should not be initialized with designated initializers.

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`
`intmax`	`intmax_t`
`uintmax`	`uintmax_t`
`size`	`size_t`
`ptrdiff`	`ptrdiff_t`
`intptr`	`intptr_t`
`uintptr`	`uintptr_t`
`int`*N*	`int`*N*`_t`
`uint`*N*	`uint`*N*`_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_inexact(x)
```

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

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

returns x's value. If the inexact flag is clear, x's value is implied to correctly represent the mathematical value of whatever operation(s) produced x. If the inexact 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. (Because the C committee has adopted two's-complement representation, 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 must represent the lower-order bits of the mathematically correct value.)

In the October 2020 meeting, WG14 conducted several straw polls about the form of the ckd_mk_* services. When a function family was suggested, the poll results were 12 in favor, one against, and 8 abstaining. When a type-generic macro was suggested, the poll results were 8 in favor, 3 against, and 10 abstaining. Since both polls indicated favor, we provided both the macro and function family.

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

```
ckd_$TYPE_t ckd_mk_$TYPE_t($TYPE value, bool inexact);
```

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

Likewise, the following macro serves the same purpose:

```
ckd_$TYPE_t ckd_mk($TYPE value, bool inexact);
```

This explicitly constructs a checked integer type given a plain integer and an inexact flag.

Checked integers can only be initialized through the functions or macro described above. They may not be initialized through an initializer list.

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` must be a pointer to any unchecked or checked integer type. If `result` points to a checked integer type, the inexact 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 resulting expression follows the usual arithmetic conversions.

Each macro's arguments will undergo default integer promotions before being passed to the macro, but will not undergo any implicit conversions, including the usual arithmetic conversions. Since checked integer types do not support implicit or explicit 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-argument 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 `<stdckdint.h>` to the list of header files in section 7.1.2 paragraph 3.

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

7.32.18 Checked Arithmetic Functions `<stdckdint.h>`

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

Update Annex B with the definitions in the following section:

Add a new section to chapter 7 after 7.23 "`_Noreturn`":

7.24 Checked Integer Arithmetic
1 The header `<stdckdint.h>` defines several macros for performing checked integer arithmetic.

7.24.1 Checked Integer Operations

Synopsis
1
```
#include <stdckdint.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 macros perform addition, subtraction, or multiplication of the mathematical values of `a` and `b`, storing the result of the operation in `*result`, (that is, `*result` is assigned the result of computing `a + b`, `a - b`, or `a * b`). Each operation is performed as if both operands were represented in a signed integer type with infinite range, and the result was then converted from this integer type to `type1`.

3 Both `type2` and `type3` shall be any integer type other than plain `char`, `bool`, or an enumeration type, and they need not be the same. `*result` shall be a modifiable lvalue of any integer type other than plain `char`, `bool`, or an enumeration type.

Recommended practice

4 It is recommended to produce a diagnostic message if `type2` or `type3` are not suitable integer types, or if `*result` is not a modifiable lvalue of a suitable integer type.

Returns
5 If these macros return `false`, the value assigned to `*result` correctly represents the mathematical result of the operation. Otherwise, these macros return `true`. In this case, `*result` is the mathematical result reduced by 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

```
ckd_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 `false`.

Supplemental Proposal

Add `<stdckdint.h>` to the list of header files in section 7.1.2 paragraph 3.

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

7.32.18 Checked Arithmetic Functions `<stdckdint.h>`

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

Update Annex B with the definitions in the following section:

Add a new section to chapter 7 after 7.23 "`_Noreturn`": The following text replaces the text suggested by the Core Proposal:

7.24 Checked Integer Arithmetic

1 The header `<stdckdint.h>` defines several macros, functions, and types for performing checked integer arithmetic.

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`

Type	Key
`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_least`*N*`_t`	`int_least`*N*
`uint_least`*N*`_t`	`uint_least`*N*
`int_fast`*N*`_t`	`int_fast`*N*
`uint_fast`*N*`_t`	`uint_fast`*N*

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 whether overflow, truncation, or misinterpretation of sign occurred when computing the integer value. The "*type*" in "`ckd_`*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`.

6. If two unchecked integer types are compatible, the analogous checked types are also compatible.

7.24.1 The `ckd_inexact` Macro

Synopsis

1

```
#include <stdckdint.h>
bool ckd_inexact(ckd_type_t x);
```

Description
2 If `ckd_type` is a checked integer type, the `ckd_inexact` macro indicates if `x` was computed using an operation that overflowed or suffered truncation or misinterpretation of sign.

Recommended practice

3 It is recommended to produce a diagnostic message if `ckd_type_t` is not a checked integer type.

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

7.24.2 The `ckd_value` Macro

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

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

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

Recommended practice

4 It is recommended to produce a diagnostic message if `ckd_type_t` is not a checked integer type.

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

7.24.3 The `ckd_mk_type_t` functions

Synopsis
1
```
#include <stdckdint.h>
[[nodiscard]] ckd_type_t ckd_mk_type_t(type value, bool inexact);
```

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

3 if the inexact flag is `true`, 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 7.24.3p3 should state:

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

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

7.24.4 The `ckd_mk` macro

Synopsis
1
```
#include <stdckdint.h>
ckd_type_t ckd_mk(type value, bool inexact);
```

Description
2 This macro constructs a checked integer type given an unchecked integer and an inexact flag.

3 If the inexact flag is `true`, 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 7.24.4p3 should state:

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

Recommended practice

4 It is recommended to produce a diagnostic message if `type` is not a suitable integer type.

Returns
5 This macro returns a checked type that represents the value indicated by `value` and the exact state indicated by `inexact`.

7.24.5 The `ckd_op` macros

Synopsis
1
```
#include <stdckdint.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 macros perform addition, subtraction, or multiplication of the mathematical values of `a` and `b`. In the first form, they store the result of the operation in `*result`, and in the second form, they return the result as a checked integer. Each operation is performed as if both operands were represented in a signed integer type with infinite range, and the result was then converted from this integer type to a particular type. For the first form, this particular type is `type1`. For the second form, this type is the type that would have been used had the operands undergone usual arithmetic conversion. (Section 6.3.1.8)

3 Both `a` and `b` shall be of any checked or unchecked integer type other than plain `char`, `bool`, or an enumeration type and their types need not be the same.

4 In the first form, `*result` shall be a modifiable lvalue and `type1` shall be any checked or unchecked integer type other than plain `char`, `bool`, or an enumeration type. In the second form, the result will be a checked integer whose type is determined by the usual arithmetic conversions.

5 In the first form, the return value indicates if an error occurred in the operation or either argument was a checked type whose inexact flag indicated a previous error. In the second form, this information is stored in the inexact flag in the return value.

Recommended practice

6 It is recommended to produce a diagnostic message if `type1`, `type2`, or `type3` are neither checked integer types nor suitable unchecked integer types.

Returns
7 The macros of the first-form return `false` if `type1` is sufficient to hold the mathematical result of the computation, which is then stored in `*result`... Otherwise, these macros return `true`.. In this case, `*result` is the mathematical result reduced by modular arithmetic on two's-complement representation with silent wrap-around on overflow. If `*result` is a checked integer type then its inexact flag will equal the macro's return value.

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

9 EXAMPLE If `a` and `b` are values of type signed int, and `result` is a signed long, then
```
ckd_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 `false`. This behavior occurs whether `result`, `a`, and `b` are checked or unchecked.

10 EXAMPLE If `a` and `b` are values of type signed int and signed long, then
```
ckd_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 clear inexact flags.

Acknowledgements

This proposal was suggested by Dr. Will Klieber.

Special thanks to Martin Sebor, Aaron Ballman, Jens Gustedt, Joseph Myers, Robert Seacord, Rajan Bhakta, 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