WG 14, N2428

Towards Integer Safety

David Svoboda
svoboda@cert.org

Date: 2019-08-16

The Problem

Integer arithmetic in C is unsafe. 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.

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 (for signed integers) or wrapping (for unsigned integers) 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.

Since 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 or wrap. 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

It is actually built with C++, and is written 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 becoming a Boost library, Robert Ramey proposed adding it 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 still will silently wrap if the mathematical solution cannot be represented by the expression type.

More information is available at:

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

Approach

The approach we take 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.

Compile-time evaluation

A solution that can be used to compute compile-time constants would be 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 is mainly addressed by the convention of capitalizing the first letter of newly-proposed functions and prepending them with underscore, as in _New_function. Developers are discouraged from defining their own functions that begin with underscore. So adding many standard functions pollutes only the namespace of standard functions, but not the namespace of user-defined functions.

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 compile-time constant expressions.

We first propose a type to represent checked integers:

```
checked_$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**
i	signed int
ui	unsigned int
l	signed long
ul	unsigned long
ll	signed long long
ull	unsigned long long
j	intmax_t
uj	uintmax_t
z	size_t
t	ptrdiff_t

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

```
bool checked_overflow(checked_$TYPE x)
```

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

```
$TYPE checked_value(checked_$TYPE 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 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:

```
checked_$TYPE_t checked_$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)

$OP	Operation
`sub`	`-` (Subtraction)
`mul`	`*` (Multiplication)

In the resulting checked type, overflow indicates whether overflow (for signed types) or wrapping (for unsigned types) 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 compile-time constant expressions, then the returned struct is also a compile-time constant expression.

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

```
checked_i_t checked_i_add(int a, int b);
```

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

Supplemental Proposal

The supplemental proposal is built 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:

```
checked_$TYPE_t make_checked_$TYPE($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).

The proposal also provides these more sophisticated functions:

```
checked_$TYPE_t checked_c$TYPE_$OP(checked_$TYPE a, checked_$TYPE b);
checked_$TYPE_t checked_c$TYPE_$TYPE_$OP(checked_$TYPE a, $TYPE b);
checked_$TYPE_t checked_$TYPE_c$TYPE_$OP($TYPE a, checked_$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:

```
checked_ci_t checked_ci_i_add(checked_i 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 a 'plain' 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:

```
checked_add(x,y)
checked_sub(x,y)
checked_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 checked_add() calls checked_i_add() if x and y are ints, checked_l_cl_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:

```
checked_mul(checked_add(a,b), checked_add(c,d))
```

The result will have an overflow bit that correctly indicates if either addition or multiplication overflowed, regardless 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 checked_add() macro, although it only considers its parameters to be either signed ints or checked signed ints.

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

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

checked_i_t {
  bool overflow;
  int value;
};

#define mk_c_i_t(x) ((checked_i_t) {false, x})

checked_i_t
checked_ci_ci_add(checked_i_t x, checked_i_t y) {
```

```
  checked_i_t result;
  result.value = 0;
  result.overflow =
    __builtin_add_overflow( x.value, y.value, &(result.value))
    || x.overflow || y.overflow;
  return result;
}

checked_i_t
checked_ci_i_add(checked_i_t x, int y) {
  return checked_ci_ci_add(x,mk_c_i_t(y));
}

checked_i_t
checked_i_ci_add(int x, checked_i_t y) {
  return checked_ci_ci_add(mk_c_i_t(x),y);
}

checked_i_t
checked_i_i_add(int x, int y) {
  return checked_ci_ci_add(mk_c_i_t(x),mk_c_i_t(y));
}


#define checked_add(x,y)                                        \
  _Generic((x),                                                 \
           checked_i_t: (_Generic((y),                          \
                      checked_i_t: checked_ci_ci_add,           \
                      int: checked_ci_i_add,                    \
                      default: NULL /* error */)),              \
           int:  (_Generic((y),                                 \
                      checked_i_t: checked_i_ci_add,            \
                      int: checked_i_i_add,                     \
                      default: NULL /* error */)),              \
           default: NULL /* error */)                           \
  (x,y)


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

Proposed Wording Changes

TODO

Acknowledgements

This proposal was suggested by Dr. Will Klieber.

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