Correct Integer Operations With Minimal Runtime Penalties

Robert Ramey, Robert Ramey Software Development

22 December 2016

1. Introduction

This library is intended as a drop-in replacement for all built-in integer types in any program which must:

- be demonstrably and verifiably correct.
- detect every user error such as input, assignment, etc.
- be efficient as possible subject to the constraints above.

1.1. Problem

Arithmetic operations in C/C++ are NOT guaranteed to yield a correct mathematical result. This feature is inherited from the early days of C. The behavior of `int`, `unsigned int` and others were designed to map closely to the underlying hardware. Computer hardware implements these types as a fixed number of bits. When the result of arithmetic operations exceeds this number of bits, the result will not be arithmetically correct. The following example illustrates just one example where this causes problems.

```
int f(int x, int y){
    // this returns an invalid result for some legal values of x and y !
    return x + y;
}
```

It is incumbent upon the C/C++ programmer to guarantee that this behavior does not result in incorrect or unexpected operation of the program. There are no language facilities which implement such a guarantee. A programmer needs to examine each expression individually to know that his program will not return an invalid result. There are a number of ways to do this. In the above instance, [INT32-C] seems to recommend the following approach:

```
int f(int x, int y){
  if (((y > 0) && (x > (INT_MAX - y)))
  || ((y < 0) && (x < (INT_MIN - y)))) {
    /* Handle error */
  }
  return x + y;
}
```

This will indeed trap the error. However, it would be tedious and laborious for a programmer to alter his code to do so. Altering code in this way for all arithmetic operations would likely render the code unreadable and add another source of potential programming errors. This approach is clearly not functional when the expression is even a little more complex as is shown in the following example.

```
int f(int x, int y, int z){
    // this returns an invalid result for some legal values of x and y !
    return x + y * z;
```

```
}
```

This example addresses only the problem of undefined/erroneous behavior related to overflow of the addition operation as applied to the type `int`. Similar problems occur with other built-in integer types such as `unsigned`, `long`, etc. And it also applies to other operations such as subtraction, multiplication etc. . C/C++ often automatically and silently converts some integer types to others in the course of implementing binary operations and similar problems occur in this case as well. Since the problems and their solution are similar, We'll confine the current discussion to just this one example.

1.2. Solution

This library implements special versions of `int`, `unsigned`, etc. which behave exactly like the original ones **except** that the results of these operations are guaranteed to be either arithmetically correct or invoke an error. Using this library, the above example would be rendered as:

```
#include <boost/safe_numeric/safe_integer.hpp>
using namespace boost::numeric;
safe<int> f(safe<int> x, safe<int> y){
  return x + y; // throw exception if correct result cannot be returned
}
```

Note

Library code in this document resides in the name space `boost::numeric`. This name space has generally been eliminated from text, code and examples in order to improve readability of the text.

The addition expression is checked at runtime or (if possible) at compile time to trap any possible errors resulting in incorrect arithmetic behavior. This will permit one to write arithmetic expressions that cannot produce an erroneous result. Instead, one and only one of the following is guaranteed to occur.

- the expression will yield the correct mathematical result
- the expression will emit a compilation error.
- the expression will invoke a runtime exception.

In other words, the **library absolutely guarantees that no arithmetic expression will yield incorrect results**.

1.3. How It Works

The library implements special versions of `int`, `unsigned`, etc. named `safe<int>`, `safe<unsigned int>` etc. These behave exactly like the underlying types **except** that expressions using these types fulfill the above guarantee. These types are meant to be "drop-in" replacements for the built-in types they are meant to replace. So things which are legal - such as assignment of a `signed` to `unsigned` value - are not trapped at compile time as they are legal C/C++ code. Instead, they are checked at runtime to trap the case where this (legal) operation would lead to an arithmetically incorrect result.

Note that the library addresses arithmetical errors generated by straightforward C/C++ expressions. Some of these arithmetic errors are defined as conforming to the C/C++ standards while others are not. So characterizing this library as addressing undefined behavior of C/C++ numeric expressions would be misleading.

Facilities particular to C++14 are employed to minimize any runtime overhead. In many cases there is no runtime overhead at all. In other cases, a program using the library can be slightly altered to achieve the above guarantee without any runtime overhead.

1.4. Additional Features

Operation of safe types is determined by template parameters which specify a pair of policy classes which specify the behavior for type promotion and error handling. In addition to the usage serving as a drop-in replacement for standard integer types, users of the library can:

- Select or define an exception policy class to specify handling of exceptions.
  - throw exception on runtime, trap at compile time.
  - trap at compiler time all operations which might fail at runtime.
  - specify custom functions which should be called at runtime
- Select or define a promotion policy class to alter the C/C++ type promotion rules. This can be used to
  - use C/C++ native type promotion rules so that, except for throwing/trapping of exceptions on operations resulting in incorrect arithmetic behavior, programs will operate identically when using/not using safe types.
  - replace C/C++ native promotion rules with ones which are arithmetically equivalent but minimize the need for runtime checking of arithmetic results.
  - replace C/C++ native promotion rules with ones which emulate other machine architectures. This is designed to permit the testing of C/C++ code destined to be run on another machine on one's development platform. Such a situation often occurs while developing code for embedded systems.
- Enforce of other program requirements using ranged integer types. The library includes the types `safe_range(Min, Max)` and `safe_literal(N)`. These types can be used to improve program correctness and performance.

1.5. Requirements

This library is composed entirely of C++ Headers. It requires a compiler compatible with the C++14 standard.

The following Boost Libraries must be installed in order to use this library

- MPL
- Integer
- Config
- Concept Checking
- Tribool
- Enable_if

The Safe Numerics library is delivered with an exhaustive suite of test programs. Users who choose to run this test suite will also need to install the Boost.Preprocessor library.

1.6. Scope

This library currently applies only to built-in integer types. Analogous issues arise for floating point types but they are not currently addressed by this version of the library. User or library defined types such as arbitrary precision integers can also have this problem. Extension of this library to these other types is not currently under development but may be addressed in the future. This is one reason why the library name is "safe numeric" rather than "safe integer" library.

2. Eliminating Runtime Penalty

Up until now, we've focused on detecting when incorrect results are produced and handling these occurrences either by throwing an exception or invoking some designated function. We've achieved our goal of detecting and handling arithmetically incorrect behavior - but at what cost. It is a fact that many C++ programmers will find this trade-off unacceptable. So the question arises as to how we might minimize or eliminate this runtime penalty.

The first step is to determine what parts of a program might invoke exceptions. The following program is similar to previous examples but uses a special exception policy: `trap_exception`.

```
#include <iostream>

#include "../include/safe_integer.hpp"
#include "../include/exception.hpp" // include exception policies

using safe_t = boost::numeric::safe<
    int,
    boost::numeric::native,
    boost::numeric::trap_exception  // note use of "trap_exception" policy!
>;

int main(int argc, const char * argv[]){
    std::cout << "example 81:\n";
    safe_t x(INT_MAX);
    safe_t y(2);
    safe_t z = x + y; // will fail to compile !
    return 0;
}
```

Now, any expression which ***might*** fail at runtime is flagged with a compile time error. There is no longer any need for `try/catch blocks`. Since this program does not compile, the **library absolutely guarantees that no arithmetic expression will yield incorrect results** . This is our original goal. Now all we need to do is make the program work. There are a couple of ways to do this.

2.1. Using Automatic Type Promotion

The C++ standard describes how binary operations on different integer types are handled. Here is a simplified version of the rules:

- promote any operand smaller than `int` to an `int` or `unsigned int`.
- if the signed operand is larger than the signed one, the result will be signed, otherwise the result will be unsigned.
- expand the smaller operand to the size of the larger one

So the result of the sum of two integer types may result in another integer type. If the values are large, the result can exceed the size that the resulting integer type can hold. This is what we call "overflow". The C/C++ standard characterises this as undefined behavior and leaves to compiler implementors the decision as to how such a situation will be handled. Usually, this means just truncating the result to fit into the result type - which sometimes will make the result arithmetically incorrect. However, depending on the compiler, compile time switch settings, the such case may result in some sort of run time exception.

The complete signature for a safe integer type is:

```
template <
```

```
    class T,                  // underlying integer type
    class P = native,         // type promotion policy class
    class E = throw_exception // error handling policy class
>
safe;
```

The standard C++ type promotion rules are consistent with the default `native` type promotion policy. Up until now, we've focused on detecting when this happens and invoking an interrupt or other kind of error handler.

But now we look at another option. Using the `automatic` type promotion policy, we can change the rules of C++ arithmetic for safe types to something like the following:

- for any C++ numeric type, we know from `std::numeric_limits` what the maximum and minimum values that a variable can be - this defines a closed interval.
- For any binary operation on these types, we can calculate the interval of the result at compile time.
- From this interval we can select a new type which can be guaranteed to hold the result and use this for the calculation. This is more or less equivalent to the following code:

  ```
  int x, y;
  int z = x + y               // could overflow

  int x, y;
  long z = (long)x + (long)y; // can never overflow
  ```

  One could do this by editing his code manually, but such a task would be tedious, error prone, and leave the resulting code hard to read and verify. Using the `automatic` type promotion policy will achieve the equivalent result without these problems
- Since the result type is guaranteed to hold the result, there is no need to check for errors - they can't happen !!! The usage of `trap_exception` exception policy enforces this guarantee
- Since there can be no errors, there is no need for `try/catch` blocks.
- The only runtime error checking we need to do is when safe values are initialized or assigned from values which are "too large". These are infrequent occurrences which generally have little or no impact on program running time. And many times, one can make small adjustments in selecting the types in order to eliminate all runtime penalties.

In short, given a binary operation, we silently promote the types of the operands to a wider result type so the result cannot overflow. This is a fundamental departure from the C++ Standard behavior.

If the interval of the result cannot be contained in the largest type that the machine can handle (usually 64 bits these days), the largest available integer type with the correct result sign is used. So even with our "automatic" type promotion scheme, it's still possible to overflow. In this case, and only this case, is runtime error checking code generated. Depending on the application, it should be rare to generate error checking code, and even more rare to actually invoke it. Any such instances are detected by the `trap_exception` exception policy.

This small example illustrates how to use automatic type promotion to eliminate all runtime penalty.

```
#include <iostream>

#include "../include/safe_integer.hpp"
#include "../include/exception.hpp"
#include "../include/automatic.hpp"
#include "safe_format.hpp" // prints out range and value of any type
```

```
using safe_t = boost::numeric::safe<
    int,
    boost::numeric::automatic, // note use of "automatic" policy!!!
    boost::numeric::trap_exception
>;

int main(int argc, const char * argv[]){
    std::cout << "example 82:\n";
    safe_t x(INT_MAX);
    safe_t y = 2;
    std::cout << "x = " << safe_format(x) << std::endl;
    std::cout << "y = " << safe_format(y) << std::endl;
    std::cout << "z = " << safe_format(x + y) << std::endl;
    return 0;
}
```

The above program produces the following output:

```
example 82:
x = <int>[-2147483648,2147483647] = 2147483647
y = <int>[-2147483648,2147483647] = 2
z = <long>[-4294967296,4294967294] = 2147483649
```

The output uses a custom output manipulator for safe types to display the underlying type and its range as well as current value. Note that:

- the `automatic` type promotion policy has rendered the result of the some of two `integers` as a `long` type.
- our program compiles without error - even when using the `trap_exception` exception policy
- We do not need to use `try/catch` idiom to handle arithmetic errors - we will have none.
- We only needed to change two lines of code to achieve our goal

2.2. Using safe_range

Instead of relying on automatic type promotion, we can just create our own types in such a way that we know they won't overflow. In the example below, we presume we know that the values we want to work with fall in the range [-24,82]. So we "know" the program will always result in a correct result. But since we trust no one, and since the program could change and the expressions be replaced with other ones, we'll still use the `trap_exception` exception policy to verify at compile time that what we "know" to be true is in fact true.

```
#include <iostream>

#include "../include/safe_range.hpp"
#include "../include/safe_literal.hpp"
#include "../include/exception.hpp"
#include "../include/native.hpp"
#include "safe_format.hpp" // prints out range and value of any type

using namespace boost::numeric; // for safe_literal

// create a type for holding small integers.  We "know" that C++
// type promotion rules will work such that addition will never
// overflow. If we change the program to break this, the usage
```

```
// of the trap_exception promotion policy will prevent compilation.
using safe_t = safe_signed_range<
    -24,
    82,
    native,         // C++ type promotion rules work OK for this example
    trap_exception  // catch problems at compile time
>;

int main(int argc, const char * argv[]){
    std::cout << "example 83:\n";
    // the following would result in a compile time error
    // since the sum of x and y wouldn't be in the legal
    // range for z.
    // const safe_signed_literal<20> x;
    const safe_signed_literal<10> x;    // no problem
    const safe_signed_literal<67> y;

    const safe_t z = x + y;
    std::cout << "x = " << safe_format(x) << std::endl;
    std::cout << "y = " << safe_format(y) << std::endl;
    std::cout << "z = " << safe_format(z) << std::endl;
    return 0;
}
```

- `safe_signed_range` defines a type which is limited to the indicated range. Out of range assignments will be detected at compile time if possible (as in this case) or at run time if necessary.
- `safe_literal` defines a constant with a specific value. Defining constants in this way enables the library to correctly anticipate the range of the results of arithmetic expressions.
- The usage of "trap exception" will mean that any assignment to z which could be outside the legal range will result in a compile time error.
- So if this program compiles, it's guaranteed to return a valid result.

This program produces the following run time output.

```
example 83:
x = <signed char>[10,10] = 10
y = <signed char>[67,67] = 67
z = <signed char>[-24,82] = 77
```

2.3. Mixing Approaches

For purposes of exposition, we've divided the discussion of how to eliminate runtime penalties by the different approaches available. A realistic program would likely include all techniques mentioned above. Consider the following:

```
#include <stdexcept>
#include <iostream>

#include "../include/safe_range.hpp"
#include "../include/automatic.hpp"
#include "../include/exception.hpp"

#include "safe_format.hpp" // prints out range and value of any type

using namespace boost::numeric;
```

```
using safe_t = safe_signed_range<
    -24,
    82,
    automatic,
    trap_exception
>;

// define variables use for input
using input_safe_t = safe_signed_range<
    -24,
    82,
    automatic, // we don't need automatic in this case
    throw_exception // these variables need to
>;

// function arguments can never be outside of limits
auto f(const safe_t & x, const safe_t & y){
    auto z = x + y;  // we know that this cannot fail
    std::cout << "z = " << safe_format(z) << std::endl;
    std::cout << "(x + y) = " << safe_format(x + y) << std::endl;
    std::cout << "(x - y) = " << safe_format(x - y) << std::endl;
    return z;
}

int main(int argc, const char * argv[]){
    std::cout << "example 84:\n";
    input_safe_t x, y;
    try{
        std::cin >> x >> y; // read varibles, maybe throw exception
    }
    catch(const std::exception & e){
        // none of the above should trap. Mark failure if they do
        std::cout << e.what() << std::endl;
        return 1;
    }
    std::cout << "x" << safe_format(x) << std::endl;
    std::cout << "y" << safe_format(y) << std::endl;
    std::cout << safe_format(f(x, y)) << std::endl;
    return 0;
}
```

- As before, we define a type `safe_t` to reflect our view of legal values for this program. This uses `automatic` type promotion policy as well as `trap_exception` exception policy to enforce elimination of runtime penalties.

- The function f accepts only arguments of type `safe_t` so there is no need to check the input values. This performs the functionality of ***programming by contract*** with no runtime cost.

- In addition, we define `input_safe_t` to be used when reading variables from the program console. Clearly, these can only be checked at runtime so they use the throw_exception policy. When variables are read from the console they are checked for legal values. We need no ad hoc code to do this, as these types are guaranteed to contain legal values and will throw an exception when this guarantee is violated. In other words, we automatically get checking of input variables with no additional programming.

- On calling of the function f, arguments of type `input_safe_t` are converted to values of type `safe_t` . In this particular example, it can be determined at compile time that construction of an instance of a `safe_t` from an `input_safe_t` can never fail. Hence, no `try/catch` block is necessary. The usage of the `trap_exception` policy for `safe_t` types would cause a compile time error.

Here is the output from the program when values 12 and 32 are input from the console:

```
example 84:
12 32
x<signed char>[-24,82] = 12
y<signed char>[-24,82] = 32
z = <short>[-48,164] = 44
(x + y) = <short>[-48,164] = 44
(x - y) = <short>[-106,106] = -20
<short>[-48,164] = 44
```

3. Background

This library is a re-implementation of the facilities provided by David LeBlanc's SafeInt Library using C++14 and the Boost Libraries. I found this library very well done in every way. My main usage was to run unit tests for my embedded systems projects on my PC. Still, I had a few issues.

- It was a lot of code in one header - 6400 lines. Very unwieldy to understand, modify and maintain.
- I couldn't find separate documentation other than that in the header file.
- It didn't use Boost conventions for naming.
- It required porting to different compilers.
- It had a very long license associated with it.
- I could find no test suite for the library.

This version addresses these issues. It exploits Boost facilities such as template metaprogramming to reduce the number of lines of source code to approximately 4700. It exploits the Boost Preprocessor Library to generate exhaustive tests.

All concepts, types and functions documented are declared in the name space `boost::numeric`.

4. Library Internals

This library should compile and run correctly on any conforming C++14 compiler.

The Safe Numerics library is implemented in terms of some more fundamental software components described here. It is not necessary to know about these components to use the library. This information has been included to help those who want to understand how the library works so they can extend it, correct bugs in it, or understand it's limitations. These components are also interesting in their own right. For all these reasons, they are documented here. In general terms, the library works in the following manner:

- All unary/binary expressions where one of the operands is a "safe" type are Overloaded. These overloads are declared and defined in the header file "safe_integer.hpp". SFINAE - "Substitution Failure Is Not An Error and `std::enable_if` are key features of C++ used to define these overloads in a correct manner.
- Each overloaded operation implements the following:
  - Retrieve range of values for each operand of type T from `std::numeric_limits<T>::min()` and `std::numeric_limits<T>::max()`.
  - Given the ranges of the operands, determine the range of the result of the operation using interval arithmetic. This is implemented in the "interval.hpp" header file using constexpr facility of C++14.

- if the range of the result type includes the range of the result of the operation, no run time checking of the result is necessary. So the operation reduces to the original built-in C/C++ operation.
- Otherwise, the operation is implemented as a "checked integer operation" at run time. This operation returns a variant which will contain either a correct result or an exception enum indicating why a correct result could not be obtained. The variant object is implemented in the header file "checked_result.hpp" and the checked operations are implemented in "checked.hpp".
- if a valid result has been obtained, it is passed to the caller.
- Otherwise, an exception is invoked.

5. Rationale and FAQ

5.1. Is this really necessary? If I'm writing the program with the requisite care and competence, problems noted in the introduction will never arise. Should they arise, they should be fixed "at the source" and not with a "band aid" to cover up bad practice.

This surprised me when it was first raised. But some of the feedback I've received makes me thing that it's a widely held view. The best answer is to consider the examples in the Tutorials and Motivating Examples section of the library documentation.

5.2. Can safe types be used as drop-in replacement for built-in types?

Almost. Replacing all built-in types with their safe counterparts should result in a program that will compile and run as expected. In some cases compile time errors will occur and adjustments to the source code will be required. Typically these will result in code which is more correct.

5.3. Why is Boost.Convert not used?

I couldn't figure out how to use it from the documentation.

5.4. Why is the library named "safe ..." rather than something like "checked ..." ?

I used "safe" in large part this is what has been used by other similar libraries. Maybe a better word might have been "correct" but that would raise similar concerns. I'm not inclined to change this. I've tried to make it clear in the documentation what the problem that the library addressed is.

5.5. Given that the library is called "numerics" why is floating point arithmetic not addressed?

Actually, I believe that this can/should be applied to any type T which satisfies the type requirement "Numeric" type as defined in the documentation. So there should be specializations `safe<float>` and related types as well as new types like `safe<fixed_decimal>` etc. But the current version of the library only addresses integer types. Hopefully the library will evolve to match the promise implied by its name.

5.6. Isn't putting a defensive check just before any potential undefined behavior is often considered a bad practice?

By whom? Is leaving code which can produce incorrect results better? Note that the documentation contains references to various sources which recommend exactly this approach to mitigate the problems created by this C/C++ behavior. See [Seacord]

5.7. It looks like the implementation presumes two's complement arithmetic at the hardware level. So this library is not portable - correct? What about other hardware architectures?

As far as is known as of this writing, the library does not presume that the underlying hardware is two's compliment. However, this has yet to be verified in a rigorous way.

5.8. Why do you specialize `numeric_limits` for "safe" types? Do you need it?

`safe<T>` behaves like a "number" just as int does. It has max, min, etc Any code which uses numeric limits to test a type `T` should works with `safe<T>`. `safe<T>` is a drop-in replacement for `T` so it has to implement all the operations.

5.9. According to C/C++ standards, unsigned integers cannot overflow - they are modular integers which "wrap around". Yet the safe numerics library detects and traps this behavior as errors. Why is that?

The guiding purpose of the library is to trap incorrect arithmetic behavior - not just undefined behavior. Although a savvy user may understand and keep present in his mind that an unsigned integer is really a modular type, the plain reading of an arithmetic expression conveys the idea that all operands are integers. Also in many cases, unsigned integers are used in cases where modular arithmetic is not intended, such as array indexes. Finally, the modulus for such an integer would vary depending upon the machine architecture. For these reasons, in the context of this library, an unsigned integer is considered to a representation of a subset of integers. Note that this decision is consistent with [INT30-C], “Ensure that unsigned integer operations do not wrap” in the CERT C Secure Coding Standard [Seacord].

5.10. Why does the library require C++14?

The original version of the library used C++11. Feedback from CPPCon, Boost Library Incubator and Boost developer's mailing list convinced me that I had to address the issue of run-time penalty much more seriously. I resolved to eliminate or minimize it. This led to more elaborate meta-programming. But this wasn't enough. It became apparent that the only way to really minimize run-time penalty was to implement compile-time integer range arithmetic - a pretty elaborate sub library. By doing range arithmetic at compiler-time, I could skip runtime checking on many/most integer operations. C++11 constexpr wasn't quite enough to do the job. C++14 constexpr can do the job. The library currently relies very heavily on C++14 constexpr. I think that those who delve into the library will be very surprised at the extent that minor changes in user code can produce guaranteed correct integer code with zero run-time penalty.

5.11. This is a C++ library - yet you refer to C/C++. Which is it?

C++ has evolved way beyond the original C language. But C++ it's still (mostly) compatible with C. So most C programs can be compiled with a C++ compiler. The problems of incorrect arithmetic afflict both C and C++. Suppose we have a legacy C program designed for some embedded system.

- Replace all `int` declarations with `int16_t` and all `long` declarations with `int32_t`.
- Create a file containing something like the following and include it at the beginning of every source file.

```
#ifdef TEST
// using C++ on test platform
#include  <cstdint>
#include <safe_integer.hpp>
#include <cpp.hpp>
using pic16_promotion = boost::numeric::cpp<
    8,  // char
    8,  // short
    8,  // int
    16, // long
    32  // long long
>;
// define safe types used desktop version of the program.
template <typename T> // T is char, int, etc data type
using safe_t = boost::numeric::safe<
    T,
    pic16_promotion,
    boost::numeric::throw_exception // use for compiling and running tests
>;
```

```
typedef safe_t<std::int16_t> int16_t;
typedef safe_t<std::int32_t> int32_t;
#else
/* using C on embedded platform */
typedef int int16_t;
typedef long int32_t;
#endif
```

- Compile tests on the desktop with a C++14 compiler and with the macro TEST defined.
- Run the tests and change code to address any thrown exceptions.

This example illustrates how this library, implemented with C++14 can be useful in the development of correct code for programs written in C.

6. Current Status

The library is currently in the Boost Review Queue. The proposal submission can be found in the Boost Library Incubator

- The library is currently limited to integers.
- Conversions to safe integer types from floating point types is not explicitly addressed.
- Note that standard library stream conversion functions such as `strtoi` etc. DO check for valid input and throw the exception `std::out_of_range` if the string cannot be converted to the specified integer type. In other words, `strtoi` already contains some of the functionality that `safe<int>` provides.
- Although care was taking to make the library portable, it's likely that at least some parts of the implementation - particularly `checked` arithmetic - depend upon two's complement representation of integers. Hence the library is probably not currently portable to other architectures.
- Currently the library permits a `safe<int>` value to be uninitialized. This supports the goal of "drop-in" replacement of C++/C built-in types with safe counter parts. On the other hand, this breaks the "always valid" guarantee.
- The library is not quite a "drop-in" replacement for all built-in integer types. In particular, C/C++ implements implicit conversions and promotions between certain integer types which are not captured by the operation overloads used to implement the library. In practice these case are few and can be addressed with minor changes to the user program to avoid these silent implicit conversions.

7. Acknowledgements

This library would never have been created without inspiration, collaboration and constructive criticism from multiple sources.

David LaBlanc	This library is inspired by David LeBlanc's SafeInt Library . I found this library very well done in every way and useful in my embedded systems work. This motivated me to take to the "next level".
Andrzej Krzemienski	Andrzej Commented and reviewed the library as it was originally posted on the Boost Library Incubator. The the consequent back and forth motivated me to invest more effort in developing documentation and examples to justify the utility, indeed the necessity, for this library. He also noted many errors in code, documentation, and tests. Without his interested and effort, I do not believe the library would have progressed beyond it's initial stages.

Boost — As always, the Boost Developer's mailing list has been the source of many useful observations from potential users and constructive criticism from very knowledgeable developers.

8. Bibliography

[coker] Zack Coker. Samir Hasan. Jeffrey Overbey. Munawar Hafiz. Christian Kästner. *Integers In C: An Open Invitation To Security Attacks?* . JTC1/SC22/WG21 - The C++ Standards Committee - ISOCPP . January 15, 2012. Coker

[crowl1] Lawrence Crowl. *C++ Binary Fixed-Point Arithmetic* . JTC1/SC22/WG21 - The C++ Standards Committee - ISOCPP . January 15, 2012. Crowl

[crowl2] Lawrence Crowl. Thorsten Ottosen. *Proposal to add Contract Programming to C++* . WG21/N1962 and J16/06-0032 - The C++ Standards Committee - ISOCPP . February 25, 2006. Crowl & Ottosen

[dietz] Will Dietz. Peng Li. John Regehr. Vikram Adve. *Understanding Integer Overflow in C/C++* . Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland . June 2012. Dietz

[garcia] J. Daniel Garcia. *C++ language support for contract programming* . WG21/N4293 - The C++ Standards Committee - ISOCPP . December 23, 2014. Garcia

[katz] Omer Katz. *SafeInt code proposal* . Boost Developer's List . Katz

[keaton] David Keaton, Thomas Plum, Robert C. Seacord, David Svoboda, Alex Volkovitsky, and Timothy Wilson. *As-if Infinitely Ranged Integer Model* . Software Engineering Institute . CMU/SEI-2009-TN-023.

[leblanc1] David LeBlanc. *Integer Handling with the C++ SafeInt Class* . Microsoft Developer Network . January 7, 2004. LeBlanc

[leblanc2] David LeBlanc. *SafeInt* . CodePlex . Dec 3, 2014. LeBlanc

[lions] Jacques-Louis Lions. *Ariane 501 Inquiry Board report* . Wikisource . July 19, 1996. Lions

[matthews] Hubert Matthews. *CheckedInt: A Policy-Based Range-Checked Integer* . Overload Journal #58 . December 2003. Matthews

[mouawad] Jad Mouawad. *F.A.A Orders Fix for Possible Power Loss in Boeing 787* . New York Times. April 30, 2015.

[plakosh] Daniel Plakosh. *Safe Integer Operations* . U.S. Department of Homeland Security . May 10, 2013. Plakosh

[seacord1] Robert C. Seacord. *Secure Coding in C and C++* . 2nd Edition. Addison-Wesley Professional. April 12, 2013. 978-0321822130. Seacord

[seacord2] Robert C. Seacord. *INT30-C. Ensure that operations on unsigned integers do not wrap* . Software Engineering Institute, Carnegie Mellon University . August 17, 2014. INT30-C

[seacord3] Robert C. Seacord. *INT32-C. Ensure that operations on signed integers do not result in overflow* . Software Engineering Institute, Carnegie Mellon University . August 17, 2014. INT32-C

[stroustrup] Bjarn Stroustrup. *The C++ Programming Language Fourth Edition*. Addison-Wesley . Copyright © 2014 by Pearson Education, Inc.. January 15, 2012. Stroustrup

[forum] Forum Posts. *C++ Binary Fixed-Point Arithmetic* . ISO C++ Standard Future Proposals . Forum