Chapter 1. Safe Numerics

Robert Ramey

Table of Contents

1. Introduction

All data types, type requirements, function and meta function names are found in the name space boost::numeric . In order to make this document more readable, we have omitted this name space qualifier.

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 avoid clutter.

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++ 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 this problem.

```
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 up 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 do this. They have to be explicitly addressed in the program code. There are a number of ways to do this. See[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 do alter his code to do. 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;
}
```

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>

safe<int> f(safe<int> x, safe<int> y){
  return x + y; // throw exception if correct result cannot be returned
}
```

The addition expression is checked at runtime or (if possible) at compile time to trap any possible errors resulting from 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. Implementation

All facilities modern C++ are employed to minimize runtime overhead required to make this guarantee. In many cases there is no runtime overhead at all. In other cases, small changes in the program are required to eliminate the runtime overhead. The library implements special versions of int, unsigned, etc. named `safe<int>`, `safe<unsigned int>` etc. These behave exactly like the original ones 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 assigning an signed to unsigned value are not trapped at compile time - as they are legal C/C++ code - but rather 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 C/C++ standard while others are not. So characterizing this library as addressing undefined behavior of C/C++ numeric expressions is misleading.

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 or 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++ type promotion rules. This can be used to
  - use C++ native type promotion rules so that, except throwing/trapping of exceptions, programs will operate identically when using/not using safe types.
  - replace C++ native promotion rules with ones which are arithmetically equivalent but minimize the need for runtime checking of arithmetic results.
  - replace C++ native promotion rules with ones which emulate other machine architectures. This is designed to permit the testing of 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 types safe_..._range<Min, Max> and safe_literal(...). 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

In order to run the test suite, the following the Boost preprocessor library is also required.

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. Tutorial and Motivating Examples

2.1. Arithmetic Expressions Can Yield Incorrect Results.

When some operation results in a result which exceeds the capacity of a data variable to hold it, the result is undefined. This is called "overflow". Since word size can differ between machines, code which produces correct results in one set of circumstances may fail when re-compiled on a machine with different hardware. When this occurs, Most C++ compilers will continue to execute with no indication that the results are wrong. It is the programmer's responsibility to ensure such undefined behavior is avoided.

This program demonstrates this problem. The solution is to replace instances of `char` type with `safe<char>` type.

```
#include <cassert>
#include <exception>
#include <iostream>
#include <cstdint>

#include "../include/safe_integer.hpp"

int main(int argc, const char * argv[]){
    std::cout << "example 1:";
    std::cout << "undetected erroneous expression evaluation" << std::endl;
    std::cout << "Not using safe numerics" << std::endl;
    // problem: arithmetic operations can yield incorrect results.
    try{
        std::int8_t x = 127;
        std::int8_t y = 2;
        std::int8_t z;
        // this produces an invalid result !
        z = x + y;
        // but assert fails to detect it since C++ implicitly
        // converts variables to int before evaluating he expression!
        // assert(z == x + y);
        std::cout << static_cast<int>(z) << " != " << x + y << std::endl;
        std::cout << "error NOT detected!" << std::endl;
    }
    catch(std::exception){
        std::cout << "error detected!" << std::endl;
    }
    // solution: replace std::int8_t with safe<std::int8_t>
```

```
    std::cout << "Using safe numerics" << std::endl;
    try{
        using namespace boost::numeric;
        safe<std::int8_t> x = 127;
        safe<std::int8_t> y = 2;
        safe<std::int8_t> z;
        // rather than producing and invalid result an exception is thrown
        z = x + y;
    }
    catch(std::exception & e){
        // which can catch here
        std::cout << e.what() << std::endl;
    }
    return 0;
}
```

Note that I've used `char` types in this example to make the problem and solution easier to see. The exact same example could have been done with `int` types albeit with different values.

2.2. Arithmetic Operations can Overflow Silently

A variation of the above is when a value is incremented/decremented beyond it's domain. This is a common problem with for loops.

```
#include <cassert>
#include <exception>
#include <iostream>

#include "../include/safe_integer.hpp"

int main(int argc, const char * argv[]){
    std::cout << "example 2:";
    std::cout << "undetected overflow in data type" << std::endl;
    // problem: undetected overflow
    std::cout << "Not using safe numerics" << std::endl;
    try{
        int x = INT_MAX;
        // the following silently produces an incorrect result
        ++x;
        std::cout << x << " != " << INT_MAX << " + 1" << std::endl;
        std::cout << "error NOT detected!" << std::endl;
    }
    catch(std::exception){
        std::cout << "error detected!" << std::endl;
    }
    // solution: replace int with safe<int>
    std::cout << "Using safe numerics" << std::endl;
    try{
        using namespace boost::numeric;
        safe<int> x = INT_MAX;
        // throws exception when result is past maximum possible
        ++x;
        assert(false); // never arrive here
    }
    catch(std::exception & e){
        std::cout << e.what() << std::endl;
        std::cout << "error detected!" << std::endl;
    }
    return 0;
```

```
}
```

When variables of unsigned integer type are decremented below zero, they "roll over" to the highest possible unsigned version of that integer type. This is a common problem which is generally never detected.

2.3. Implicit Conversions Change Data Values

A simple assignment or arithmetic expression will convert all the terms to the same type. Sometimes this can silently change values. For example, when a signed data variable contains a negative type, assigning to a unsigned type will be permitted by any C/C++ compiler but will be treated as large unsigned value. Most modern compilers will emit a compile time warning when this conversion is performed. The user may then decide to change some data types or apply a `static_cast`. This is less than satisfactory for two reasons:

- It may be unwieldy to change all the types to signed or unsigned.
- We may believe that our signed type will never contain a negative value. `static_cast` changes the data type - not the data value. If we ignore the any compiler warnings or use a `static_cast` to suppress them, we'll fail to detect a program error when it is committed. This is aways a risk with casts.

This solution is simple, Just replace instances of the `int` with `safe<int>`.

```
#include <exception>
#include <iostream>

#include "../include/safe_integer.hpp"

int main(int argc, const char * argv[]){
    std::cout << "example 4: ";
    std::cout << "implicit conversions change data values" << std::endl;
    std::cout << "Not using safe numerics" << std::endl;
    // problem: implicit conversions change data values
    try{
        int x = -1000;
        // the following silently produces an incorrect result
        char y = x;
        std::cout << x << " != " << (int)y << std::endl;
        std::cout << "error NOT detected!" << std::endl;
    }
    catch(std::exception){
        std::cout << "error detected!" << std::endl; // never arrive here
    }
    // solution: replace int with safe<int> and char with safe<char>
    std::cout << "Using safe numerics" << std::endl;
    try{
        using namespace boost::numeric;
        safe<int> x = -1000;
        // throws exception when conversion change data value
        safe<char> y1(x);
        safe<char> y2 = x;
        safe<char> y3 = {x};
        std::cout << "error NOT detected!" << std::endl;
    }
    catch(std::exception & e){
        std::cout << e.what() << std::endl;
        std::cout << "error detected!" << std::endl;
    }
    return 0;
```

```
}
```

2.4. Mixing Data Types Can Create Subtle Errors

C++ contains signed and unsigned integer types. In spite of their names, they function differently which often produces surprising results for some operands. Program errors from this behavior can be exceedingly difficult to find. This has lead to recommendations of various ad hoc "rules" to avoid these problems. It's not always easy to apply these "rules" to existing code without creating even more bugs. Here is a typical example of this problem:

```
#include <iostream>
#include <cstdint>

#include "../include/safe_integer.hpp"
#include "../include/cpp.hpp"

using namespace std;
using namespace boost::numeric;

void f(const unsigned int & x, const int8_t & y){
    cout << x * y << endl;
}
void safe_f(
    const safe<unsigned int> & x,
    const safe<int8_t> & y
){
    cout << x * y << endl;
}

int main(){
    cout << "example 10: ";
    cout << "mixing types produces surprising results" << endl;
    try {
        std::cout << "Not using safe numerics" << std::endl;
        // problem: arithmetic operations can yield incorrect results.
        f(100, 100);  // works as expected
        f(100, -100); // wrong result - unnoticed
    }
    catch(std::exception){
        // never arrive here
        std::cout << "error detected!" << std::endl;
    }
    try {
        // solution: use safe types
        std::cout << "Using safe numerics" << std::endl;
        safe_f(100, 100);  // works as expected
        safe_f(100, -100); // throw error
    }
    catch(const exception & e){
        cout << "detected error:" << e.what() << endl;;
    }
    return 0;
}
```

Here is the output of the above program:

```
example 10: mixing types produces surprising results
```

```
Not using safe numerics
10000
4294957296
Using safe numerics
10000
detected error:converted negative value to unsigned
```

This solution is simple, Just replace instances of the `int`with `safe<int>`.

2.5. Array Index Value Can Exceed Array Limits

Using an intrinsic C++ array, it's very easy to exceed array limits. This can fail to be detected when it occurs and create bugs which are hard to find. There are several ways to address this, but one of the simplest would be to use safe_unsigned_range;

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

#include "../include/safe_range.hpp"

void detected_msg(bool detected){
    std::cout << (detected ? "error detected!" : "error NOT detected! ") << std::endl;
}

int main(int argc, const char * argv[]){
    // problem: array index values can exceed array bounds
    std::cout << "example 5: ";
    std::cout << "array index values can exceed array bounds" << std::endl;
    std::cout << "Not using safe numerics" << std::endl;
    std::array<int, 37> i_array;

    // unsigned int i_index = 43;
    // the following corrupts memory.
    // This may or may not be detected at run time.
    // i_array[i_index] = 84; // comment this out so it can be tested!
    std::cout << "error NOT detected!" << std::endl;

    // solution: replace unsigned array index with safe_unsigned_range
    std::cout << "Using safe numerics" << std::endl;
    try{
        using namespace boost::numeric;
        using i_index_t = safe_unsigned_range<0, i_array.size() - 1>;
        i_index_t i_index;
        i_index = 36; // this works fine
        i_array[i_index] = 84;
        i_index = 43; // throw exception here!
        std::cout << "error NOT detected!" << std::endl; // so we never arrive here
    }
    catch(std::exception & e){
        std::cout << e.what() << std::endl;
        std::cout << "error detected!" << std::endl;
    }
    return 0;
}
```

Collections like standard arrays, vectors do array index checking in some function calls and not in others so this may not be the best example. However it does illustrate the usage of `safe_range<T>` for assigning legal range to variables. This will guarantee that under no circumstances will the variable contain a value outside of the specified range.

2.6. Checking of Input Values Can Be Easily Overlooked

It's way too easy to overlook the checking of parameters received from outside the current program.

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

#include "../include/safe_integer.hpp"

int main(int argc, const char * argv[]){
    // problem: checking of externally produced value can be overlooked
    std::cout << "example 6: ";
    std::cout << "checking of externally produced value can be overlooked" << std::endl;
    std::cout << "Not using safe numerics" << std::endl;

    std::istringstream is("12317289372189 123128738921738921789 3");

    try{
        int x, y;
        is >> x >> y; // get integer values from the user
        std::cout << x << ' ' << y << std::endl;
        std::cout << "error NOT detected!" << std::endl;
    }
    catch(std::exception){
        std::cout << "error detected!" << std::endl;
    }

    // solution: asign externally retrieved values to safe equivalents
    std::cout << "Using safe numerics" << std::endl;
    {
        using namespace boost::numeric;
        safe<int> x, y;
        is.seekg(0);
        try{
            is >> x >> y; // get integer values from the user
            std::cout << x << ' ' << y << std::endl;
            std::cout << "error NOT detected!" << std::endl;
        }
        catch(std::exception & e){
            std::cout << e.what() << std::endl;
            std::cout << "error detected!" << std::endl;
        }
    }
    return 0;
}
```

Without safe integer, one will have to insert new code every time an integer variable is retrieved. This is a tedious and error prone procedure. Here we have used program input. But in fact this problem can occur with any externally produced input.

2.7. Programming by Contract is Too Slow

Programming by Contract is a highly regarded technique. There has been much written about it has been proposed as an addition to the C++ language [Garcia][Crowl & Ottosen]. It (mostly) depends upon runtime checking of parameter and object values upon entry to and exit from every function. This can slow the program down considerably which in turn undermines the main motivation for using C++ in the first place! One popular scheme for addressing this issue is to enable parameter checking only during debugging

and testing which defeats the guarantee of correctness which we are seeking here! Programming by Contract will never be accepted by programmers as long as it is associated with significant additional runtime cost.

The Safe Numerics Library has facilities which, in many cases, can check guarantee parameter requirements with little or no runtime overhead. Consider the following example:

```
#include <cassert>
#include <stdexcept>
#include <sstream>
#include <iostream>

#include "../include/safe_range.hpp"

// NOT using safe numerics - enforce program contract explicitly
// return total number of minutes
unsigned int convert(
    const unsigned int & hours,
    const unsigned int & minutes
) {
    // check that parameters are within required limits
    // invokes a runtime cost EVERYTIME the function is called
    // and the overhead of supporting an interrupt.
    // note high runtime cost!
    if(minutes > 59)
        throw std::domain_error("minutes exceeded 59");
    if(hours > 23)
        throw std::domain_error("hours exceeded 23");
    return hours * 60 + minutes;
}

// Use safe numeric to enforce program contract automatically
// define convient typenames for hours and minutes hh:mm
using hours_t = boost::numeric::safe_unsigned_range<0, 23>;
using minutes_t = boost::numeric::safe_unsigned_range<0, 59>;

// return total number of minutes
// type returned is safe_unsigned_range<0, 24*60 - 1>
auto safe_convert(const hours_t & hours, const minutes_t & minutes) {
    // no need for checking as parameters are guaranteed to be within limits
    // expression below cannot throw ! zero runtime overhead
    return hours * 60 + minutes;
}

int main(int argc, const char * argv[]){
    std::cout << "example 7: ";
    std::cout << "enforce contracts with zero runtime cost" << std::endl;
    std::cout << "Not using safe numerics" << std::endl;

    // problem: checking of externally produced value can be expensive
    try {
        convert(10, 83); // invalid parameters - detected - but at a heavy cost
    }
    catch(std::exception e){
        std::cout << "exception thrown for parameter error" << std::endl;
    }

    // solution: use safe range to restrict parameters
    std::cout << "Using safe numerics" << std::endl;
```

```
    try {
        // parameters are guarenteed to meet requirements
        hours_t hours(10);
        minutes_t minutes(83);  // interrupt thrown here
        // so the following will never throw
        safe_convert(hours, minutes);
    }
    catch(std::exception e){
        std::cout
            << "exception thrown when invalid arguments are constructed"
            << std::endl;
    }

    try {
        // parameters are guarenteed to meet requirements when
        // constructed on the stack
        safe_convert(hours_t(10), minutes_t(83));
    }
    catch(std::exception e){
        std::cout
            << "exception thrown when invalid arguments are constructed on the stack"
            << std::endl;
    }

    try {
        // parameters are guarenteed to meet requirements when
        // implicitly constructed to safe types to match function signature
        safe_convert(10, 83);
    }
    catch(std::exception e){
        std::cout
            << "exception thrown when invalid arguments are implicitly constructed"
            << std::endl;
    }

    try {
        // the following will never throw as the values meet requirements.
        const hours_t hours(10);
        const minutes_t minutes(17);

        // note zero runtime overhead once values are constructed

        // the following will never throw because it cannot be called with
        // invalid parameters
        safe_convert(hours, minutes); // zero runtime overhead

        // since safe types can be converted to their underlying unsafe types
        // we can still call an unsafe function with safe types
        convert(hours, minutes); // zero (depending on compiler) runtime overhead

        // since unsafe types can be implicitly converted to corresponding
        // safe types we can just pass the unsafe types.  checkin will occur
        // when the safe type is constructed.
        safe_convert(10, 17); // runtime cost in creating parameters

    }
    catch(std::exception e){
        std::cout << "error detected!" << std::endl;
    }
```

```
    return 0;
}
```

In the example above the function convert incurs significant runtime cost every time the function is called. By using "safe" types, this cost is moved to moment when the parameters are constructed. Depending on how the program is constructed, this may totally eliminate extraneous computations for parameter requirement type checking. In this scenario, there is no reason to suppress the checking for release mode and our program can be guaranteed to be always arithmetically correct.

3. 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 enforcing arithmetically correct behavior - but at what cost. For many C++ programmers any runtime penalty is unacceptable. Whether or not one agrees with this trade off, its a fact that many C++ programmers feel this way. So the question arises as to how we alter our program to minimize or eliminate any 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
#include "safe_format.hpp" // prints out range and value of any type

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

safe_t f(const safe_t & x, const safe_t & y){
    // each statement below will fail to compile !
    safe_t z = x + y;
    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 81:\n";
    safe_t x(INT_MAX);  // will fail to compile
    safe_t y(2);        // will fail to compile
    std::cout << "x" << safe_format(x) << std::endl;
    std::cout << "y" << safe_format(y) << std::endl;
    std::cout << "z" << safe_format(f(x, y)) << std::endl;
    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.

3.1. 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 will result in another integer type. If the values are large, they will exceed the size that the resulting integer type can hold. This is what we call "overflow". Standard C/C++ does just truncates the result to fit into the result type - which sometimes will make the result arithmetically incorrect.

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 types, 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/catuch blocks.
- The only runtime error checking we need to do is when safe values are initialized or assigned using smaller types. 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 promote the types of the operands to a larger so that the result type 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/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
>;

auto f(const safe_t & x, const safe_t & y){ // note use of "auto"
    auto z = x + y; // note change to "auto"
    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 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(f(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 = (x + y) = <long>[-4294967296,4294967294] = 2147483649
(x - y) = <long>[-4294967295,4294967295] = 2147483645
<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:

- 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

3.2. Using safe_range

Instead of relying on automatic type promotion, we can just create our own types in such a way that we known they won't overflow. In the example below, we presume we happen to know that the values we want to work with fall in the closed range of -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 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/native.hpp"
#include "../include/exception.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 operations on this type
// will never overflow. If 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
>;

auto f(const safe_t & x,    const safe_t & y){
    //safe_t z = x + y; // depending on values of x & y COULD fail
    auto z = x + y;    // due to C++ type promotion rules,
                       // we know that this cannot fail
    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 83:\n";
    // constexpr const safe_t z = 3; // fails to compile
    const safe_t x(safe_literal<2>{});
    const safe_t y = safe_literal<2>();  // to avoid runtime penalty
    std::cout << "x = " << safe_format(x) << std::endl;
    std::cout << "y = " << safe_format(y) << std::endl;
    std::cout << "z = " << safe_format(f(x, y)) << std::endl;
    return 0;
}
```

which produces the following output.

```
example 83:
x = <signed char>[-24,82] = 2
y = <signed char>[-24,82] = 2
z = (x + y) = <int>[-48,164] = 4
(x - y) = <int>[-106,106] = 0
```

```
<int>[-48,164] = 4
```

- In this example, standard C++ type promotion rules are used. These promote operands to int before invoking the addition operation. So addition operation itself won't overflow. The result of addition is another unnamed safe type guaranteed to be able to hold the some of any pair of safe types. In this example the result is a safe type based on the C++ built-in type of `short`.
- So when we try to assign the result to z we could get an error. This is because our custom `safe_t` cannot be guaranteed to hold the some of all possible pairs of `safe_t` instances. We fix the by using an "auto" for the sum.
- We now have a problem when we try to initialize our `safe_t` variable with an initial literal value. This operation could overflow at runtime. To our disappointment, our attempt to fix the problem by using `constexpr` fails. The fix for this is to use `safe_literal` to initialize safe types. safe_literal is a special safe type which wraps a constant defined at compile time. It cannot be assigned to, or changed. Subject to this restriction, it can participate in safe arithmetic operations.

We've used simple expressions in this illustration. But since binary operations on safe types result in other safe types, expressions can be made arbitrarily elaborate - just as they can be with intrinsic integer types. That is, safe integer types are drop in replacements for intrinsic integer types. We are guaranteed never to produce an incorrect result regardless of how elaborate the expression might be.

3.3. 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/safe_literal.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; // for safe_literal

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";
    try{
        input_safe_t x, y;
        std::cin >> x >> y; // read varibles, throw exception
        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;
    }
    catch(const std::exception & e){
        // none of the above should trap. Mark failure if they do
        std::cout << e.what() << std::endl;
        return false;
    }
    return 0;
}
```

- As before we define a safe_t to reflect our view of legal values for this program. This uses automatic type policy as well as trapping exception policy to enforce elimination of runtime penalties.
- 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 and assigned to safe_t variables x and y, they are checked for legal values. We need at 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, the variables of type input_safe_t are converted to variables of safe_t. This conversion invokes any needed checking. In this case there is none necessary. If checker were necessary, the usage of the trap_exception policy for safe_t types would cause a compile time error.
- The function f accepts only safe_t types so we have no need to check the input types. This performs the functionality of *programming by contract* with no runtime cost. Due to our usage of trap_exception as a policy in safe_t, any violation in our "contract" will result in a runtime 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
```

4. Notes

This library really an re-implementation the facilities provided by David LeBlanc's SafeInt Library [http://safeint.codeplex.com] using Boost and C++14 [www.boost.org]. 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 [www.boost.org] 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 [www.boost.org] 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`.

5. Type Requirements

5.1. Numeric<T>

Description

A type is Numeric if it has the properties of a number.

More specifically, a type T is Numeric if there exists specialization of `std::numeric_limits<T>`. See the documentation for standard library class numeric_limits. The standard library includes such specializations for all the primitive numeric types. Note that this concept is distinct from the C++ standard library type traits `is_integral` and `is_arithmetic`. These latter fulfill the requirement of the concept Numeric. But there are types T which fulfill this concept for which `is_arithmetic<T>::value == false`. For example see `safe_signed_integer<int>`.

Notation

`T, U, V`	A type that is a model of the Numeric
`t, u`	An object of type modeling Numeric
`os`	An object of type std::base_ostreami
`is`	An object of type std::base_istream

Associated Types

`std::numeric_limits<T>`	The numeric_limits class template provides a C++ program with information about various properties of the implementation's representation of the arithmetic types. See C++ standard 18.3.2.2.

Valid Expressions

In addition to the expressions defined in Assignable [http://www.sgi.com/tech/stl/Assignable.html] the following expressions must be valid. Any operations which result in integers which cannot be represented as some Numeric type will throw an exception.

Table 1.1. General

Expression	Return Value
`std::numeric_limits<T>.is_bounded`	true
`std::numeric_limits<T>.is_specialized`	true
`os << t`	os &i
`is >> t`	is &

Table 1.2. Unary Operators

Expression	Return Type	Semantics
`-t`	T	Invert sign
`+t`	T	unary plus - a no op
`t--`	T	post decrement
`t++`	T	post increment
`--t`	T	pre decrement
`++t`	T	pre increment
`~`	T	complement

Table 1.3. Binary Operators

Expression	Return Type	Semantics
`t - u`	V	subtract u from t
`t + u`	V	add u to t
`t * u`	V	multiply t by u
`t / u`	T	divide t by u
`t % u`	T	t modulus u
`t << u`	T	shift t left u bits
`t >> u`	T	shift t right by u bits
`t < u`	bool	true if t less than u, false otherwise
`t <= u`	bool	true if t less than or equal to u, false otherwise

Expression	Return Type	Semantics
`t > u`	bool	true if t greater than u, false otherwise
`t >= u`	bool	true if t greater than or equal to u, false otherwise
`t == u`	bool	true if t equal to u, false otherwise
`t != u`	bool	true if t not equal to u, false otherwise
`t & u`	V	and of t and u padded out max # bits in t, u
`t | u`	V	or of t and u padded out max # bits in t, u
`t ^ u`	V	exclusive or of t and u padded out max # bits in t, u
`t = u`	`T`	assign value of u to t
`t += u`	`T`	add u to t and assign to t
`t -= u`	`T`	subtract u from t and assign to t
`t *= u`	`T`	multiply t by u and assign to t
`t /= u`	`T`	divide t by u and assign to t
`t &= u`	`T`	and t with u and assign to t
`t <<= u`	T	left shift the value of t by u bits
`t >>= u`	T	right shift the value of t by u bits
`t &= u`	T	and the value of t with u and assign to t
`t |= u`	T	or the value of t with u and assign to t
`t ^= u`	T	exclusive or the value of t with u and assign to t

Header

`#include <safe_numerics/include/concepts/numeric.hpp>` [../../include/concept/numeric.hpp]

Models

`int, safe_signed_integer<int>, safe_signed_range<int>, etc.`

5.2. Integer<T>

Description

A type is fulls the requirements of an Integer if it has the properties of a integer.

More specifically, a type T is Integer if there exists specialization of `std::numeric_limits<T>` for which `std::numeric_limits<T>:: is_integer` is equal to `true`. See the documentation for standard library class numeric_limits. The standard library includes such specializations for all the primitive numeric types. Note that this concept is distinct from the C++ standard library type traits `is_integral` and `is_arithmetic`. These latter fulfill the requirement of the concept Numeric. But there are types which fulfill this concept for which `is_arithmetic<T>::value == false`. For example see `safe<int>`.

Refinement of

Numeric

Valid Expressions

In addition to the expressions defined in Numeric the following expressions must be valid.

Expression	Return Value
`std::numeric_limits<T> is_integer`	true

Header

`#include <safe_numerics/include/concepts/numeric.hpp>` [../../include/concept/numeric.hpp]

Models

`int, safe<int>, safe_unsigned_range<0, 11>, etc.`

5.3. SafeNumeric<T>

Description

This holds an arithmetic value which can be used as a replacement for built-in C++ arithmetic values. These types differ from their built-in counter parts in that the are guaranteed not to produce invalid arithmetic results.

Refinement of

Numeric

Notation

Symbol	Description
`T, U`	Types fulfilling Numeric type requirements
t, u	objects of types T, U
S, S1, S2	A type fulfilling SafeNumeric type requirements
s, s1, s2	objects of types S
op	C++ infix operator

Symbol	Description
prefix_op	C++ prefix operator
postfix_op	C++ postfix operator
assign_op	C++ assignment operator

Valid Expressions

Expression	Result Type	Description
`s op t`	unspecified S	invoke safe C++ operator op and return another SafeNumeric type.
`t op s`	unspecified S	invoke safe C++ operator op and return another SafeNumeric type.
`s1 op s2`	unspecified S	invoke safe C++ operator op and return another SafeNumeric type.
`prefix_op S`	unspecified S	invoke safe C++ operator op and return another SafeNumeric type.
`S postfix_op`	unspecified S	invoke safe C++ operator op and return another SafeNumeric type.
`s assign_op t`	S1	convert t to type S1 and assign it to s1. If the value t cannot be represented as an instance of type S1, it is an error.
`S(t)`	unspecified S	construct a instance of S from a value of type T. f the value t cannot be represented as an instance of type S1, it is an error.
`S`	S	construct a uninitialized instance of S.
`is_safe<S>`	`std::true_type` or `std::false_type`	type trait to query whether any type T fulfills the requirements for a SafeNumeric type.
`static_cast<T>(s)`	T	convert the value of s to type T. If the value of s cannot be correctly represented as a type T, it is an error. Note that implicit casting from a safe type to a built-in integer type is expressly prohibited and should invoke a compile time error.

- Result of any binary operation where one or both of the operands is a SafeNumeric type is also a SafeNumeric type.
- All the expressions in the above table are `constexpr` expressions
- Binary expressions which are not assignments require that promotion and exception policies be identical.
- Safe Numeric operators will NOT perform standard numeric conversions in order to convert to built-in types.

```
void f(int);

int main(){
    long x;
    f(x);         // OK - builtin implicit version
    safe<long> y;
```

```
    f(y);         // compile time error
    return 0;
}
```

This behavior prevents a `safe<T>` from being a "drop-in" replacement for a `T`.

Complexity Guarantees

There are no explicit complexity guarantees here. However, it would be very surprising if any implementation were to be more complex that O(0);

Invariants

The fundamental requirement of a SafeNumeric type is that implements all C++ operations permitted on it's base type in a way the prevents the return of an incorrect arithmetic result. Various implementations of this concept may handle circumstances which produce such results differently (throw exception, compile time trap, etc..) no implementation should return an arithmetically incorrect result.

Header

`#include <safe_numerics/include/concepts/safe_numeric.hpp>` [../../include/concept/exception_policy.hpp]

Models

safe<T>

safe_signed_range<-11, 11>

safe_unsigned_range<0, 11>

safe_literal<4>

5.4. PromotionPolicy<PP>

Description

In C++, arithmetic operations result in types which may or may not be the same as the constituent types. A promotion policy determines the type of the result of an arithmetic operation. For example, in the following code

```
int x;
char y;
auto z = x + y
```

the type of `z` will be an `int`. This is a consequence for the standard rules for type promotion for C/C++ arithmetic. A key feature of library permits one to specify his own type promotion rules via a PromotionPolicy class.

Notation

`PP`	A type that full fills the requirements of a PromotionPollicy
`T, U, V`	A type that is a model of the Numeric concept
`t, u, v`	An object of type modeling Numeric

Valid Expressions

Any operations which result in integers which cannot be represented as some Numeric type will throw an exception.

Expression	Return Value
`PP::addition_result<T, U>::type`	unspecified Numeric type
`PP::subtraction_result<T, U>::type`	unspecified Numeric type
`PP::multiplication_result<T, U>::type`	unspecified Numeric type
`PP::division_result<T, U>::type`	unspecified Numeric type
`PP::modulus_result<T, U>::type`	unspecified Numeric type
`PP::left_shift_result<T>::type`	unspecified Numeric type
`PP::right_shift_result<T>::type`	unspecified Numeric type
`PP::bitwise_result<T>::type`	unspecified Numeric type

Header

`#include <safe_numerics/include/concepts/promotion_policy.hpp>` [../../include/concept/promotion_policy.hpp]

Models

The library contains a number of pre-made promotion policies:

- `boost::numeric::native`

  Use the normal C/C++ expression type promotion rules.

  ```
  int x;
  char y;
  auto z = x + y; // could result in overflow
  safe<int> sx;
  auto sz = sx + y; // includes code which traps overflows at runtime
  ```

  The type of sz will be safe< type of z >.

  This policy is documented in Promotion Policies - native.

- `boost::numeric::automatic`

  Use optimizing expression type promotion rules. These rules replace the normal C/C++ type promotion rules with other rules which are designed to result in more efficient computations. Expression types are promoted to the smallest type which can be guaranteed to hold the result without overflow. If there is no such type, the result will be checked for overflow. Consider the following example:

  ```
  int x;
  char y;
  auto z = x + y; // could result in overflow
  ```

```
safe<int> sx;
auto sz = sx + y; // promotes expression type to a safe<long int> which requires no result checking
is guaranteed not to overflow.

safe_unsigned_range<1, 4> a;
safe_unsigned_range<2, 4> b;
auto c = a + b; // c will be of type safe_unsigned_range<3, 8> and cannot overflow
```

Type sz will be a SafeNumeric type which is guaranteed to hold he result of x + y. In this case that will be a long int (or perhaps a long long) depending upon the compiler and machine architecture. In this case, there will be no need for any special checking on the result and there can be no overflow.

Type of c will be a signed character as that type can be guaranteed to hold the sum so no overflow checking is done.

This policy is documented in Promotion Policies - automatic

5.5. ExceptionPolicy<EP>

Description

The exception policy specifies what is to occur when a safe operation cannot return a valid arithmetic result. A type is an ExceptionPolicy if it has functions for handling exceptional events that occur in the course of safe numeric operations.

Notation

`EP`	A type that full fills the requirements of an ExceptionPollicy
message	A const char * which refers to a text message about the cause of an exception

Valid Expressions

Any operations which result in integers which cannot be represented as some Numeric type will throw an exception.

Expression	**Return Value**
`EP::overflow_error(const char * message)`	void
`EP::underflow_error(const char * message)`	void
`EP::range_error(const char * message)`	void

Header

`#include <safe_numerics/include/concepts/exception_policy.hpp>` [../../include/concept/exception_policy.hpp]

Models

The library header `<safe_numerics/include/exception_policies.hpp>` [../../include/exception_policies.hpp]contains a number of pre-made exception policies:

- `boost::numeric::throw_exception`

  If an exceptional condition is detected at runtime throw the exception. Safe types use this exception policy as the default if no other one is specified.

- `boost::numeric::ignore_exception`

  Emulate the normal C/C++ behavior of permitting overflows, underflows etc.

- `template<void (*F)(const char *), void (*G)(const char *), void (*H)(const char *)>`

  `boost::numeric::no_exception_support`

  If you want to specify specific behavior for particular exception types, use this policy. The most likely situation is where you don't have exception support and you want to trap "exceptions" by calling your own special functions.

- `boost::numeric::trap_exception`

  Use this policy to trap at compile time any operation which would otherwise trap at runtime. Hence expressions such as i/j will trap at compile time unless j can be guaranteed to not be zero.

6. Types

6.1. safe<T, PP = boost::numeric::native, EP = boost::numeric::throw_exception>

Description

A `safe<T, PP , EP>` can be used anywhere a type T can be used. Any expression which uses this type is guaranteed to return an arithmetically correct value or trap in some way.

Notation

Symbol	Description
`T`	Underlying type from which a safe type is being derived

Associated Types

`PP`	A type which specifies the result type of an expression using safe types.
`EP`	A type containing members which are called when a correct result cannot be returned

Template Parameters

Parameter	Type Requirements	Description
`T`	Integer<T> [http://en.cppreference.com/w/cpp/types/is_integral]	The underlying type. Currently only integer types supported
`PP`	PromotionPolicy<PP>	Default value is boost::numeric::native
`EP`	Exception Policy<EP>	Default value is boost::numeric::throw_exception

See examples below.

Model of

Integer

SafeNumeric

Valid Expressions

Implements all expressions and only those expressions defined by the SafeNumeric type requirements. This, the result type of such an expression will be another safe type. The actual type of the result of such an expression will depend upon the specific promotion policy template parameter.

Header

`#include <boost/safe_numerics/safe_integer.hpp>` [../../include/safe_integer.hpp]

Examples of use

The most common usage would be safe<T> which uses the default promotion and exception policies. This type is meant to be a "drop-in" replacement of the intrinsic integer types. That is, expressions involving these types will be evaluated into result types which reflect the standard rules for evaluation of C++ expressions. Should it occur that such evaluation cannot return a correct result, an exception will be thrown.

There are two aspects of the operation of this type which can be customized with a policy. The first is the result type of an arithmetic operation. C++ defines the rules which define this result type in terms of the constituent types of the operation. Here we refer to these rules a "type promotion" rules. These rules will sometimes result in a type which cannot hold the actual arithmetic result of the operation. This is the main motivation for making this library in the first place. One way to deal with this problem is to substitute our own type promotion rules for the C++ ones.

As a Drop-in replacement for standard integer types.

The following program will throw an exception and emit a error message at runtime if any of several events result in an incorrect arithmetic type. Behavior of this program could vary according to the machine architecture in question.

```
#include <exception>
#include <iostream>
#include <boost/numeric/safe.hpp>

void f(){
    using namespace boost::numeric;
    safe<int> j;
    try {
        safe<int> i;
        std::cin >> i;  // could overflow !
        j = i * i;      // could overflow
    }
    catch(std::exception & e){
       std::cout << e.what() << std::endl;
    }
    std::cout << j;
}
```

The term "drop-in replacement" reveals the aspiration of this library. In most cases, this aspiration is realized. But in some cases it isn't and can't be. Consider:

```
void f(int);

int main(){
    long x;
    f(x);         // OK - builtin implicit version
    safe<long> y;
    f(y);         // compile time error
    return 0;
}
```

The built-in integer types are freely and silently converted to other integer types in order to permit passing of integer arguments to other function types. Our view is that this is bad practice and creates the possibility of subtle bugs. It conflicts with the purpose of the library in a fundamental way. The library specifies that these conversions are errors that are to be invoked at compile time. If one wants to switch between save and built-in types via an alias, this type of code will have to be fixed so that implicit conversions to built-in types do not occur. In our view, this will be a net improvement to the code in any case.

Guarantee correct behavior at compile time.

In some instance catching an error at run time is not sufficient. We want to be sure that the program can never fail. To achieve this, use the trap_exception exception policy rather than the default throw exception policy.

The following program will emit a compile error at any statement which might possibly result in incorrect behavior.

This is because there is no way to guarantee that the expression i * i will return an arithmetically correct result. Since we know that the program cannot compile if there is any possibility of arithmetic error, we can dispense with the exception handling used above.

```
#include <iostream>
#include <boost/numeric/safe.hpp>

void f(){
    using safe_int = safe<int, boost::numeric::native, boost::numeric::trap_exception>;
    safe_int i;
    std::cin >> i;  // could throw exception here !!!
    safe_int j;
    j = i * i; // could throw exception here !!!
}
```

Adjust type promotion rules.

Another way to avoid arithmetic errors like overflow is to promote types to larger sizes before doing the arithmetic. This can be justified by the observe

Stepping back, we can see that many of the cases of invalid arithmetic are wouldn't exist if results types were larger. So we can avoid these problems by replacing the C++ type promotion rules for expressions with our own rules. This can be done by specifying a non-default type promotion policy automatic. The policy stores the result of an expression in the smallest size that can accommodate the largest value that an expression can yield. All the work is done at compile time - checking for exceptions necessary (input is of course an exception). The following example illustrates this.

```
#include <boost/numeric/safe.hpp>
#include <iostream>
void f(){
    using safe_int = safe<int, boost::numeric::automatic, boost::numeric::throw_exception>;
    safe_int i;
    std::cin >> i; // might throw exception
    auto j = i * i; // won't ever trap - result type can hold the maximum value of i * i
```

```
    static_assert(boost::numeric::is_safe<decltype(j)>::value); // result is another safe type
    static_assert(
        std::numeric_limits<decltype(i * i)>::max() >=
        std::numeric_limits<safe_int>::max() * std::numeric_limits<safe_int>::max()
    ); // always true
}
```

6.2. safe_signed_range<MIN, MAX, PP, EP> and safe_unsigned_range<MIN, MAX, PP, EP>

Description

This type holds a signed or unsigned integer in the closed range [MIN, MAX]. A `safe_signed_range<MIN, MAX, PP, EP>` or `safe_unsigned_range<MIN, MAX, PP, EP>` can be used anywhere an arithmetic type is permitted. Any expression which uses either of these types is guaranteed to return an arithmetically correct value or trap in some way.

Notation

Symbol	Description
`MIN, MAX`	Minimum and maximum values that the range can represent.

Associated Types

`PP`	Promotion Policy. A type which specifies the result type of an expression using safe types.
`EP`	Exception Policy. A type containing members which are called when a correct result cannot be returned

Template Parameters

Parameter	Requirements	Description
`MIN`	must be non-integer literal	The minimum non-negative integer value that this type may hold
`MAX`	must be a non-negative literal	The maximum non-negative integer value that this type may hold
	MIN <= MAX	must be a valid closed range
`PP`	PromotionPolicy<PP>	Default value is boost::numeric::native
`EP`	Exception Policy<EP>	Default value is boost::numeric::throw_exception

Model of

Integer

SafeNumeric

Valid Expressions

Implements all expressions and only those expressions defined by the SafeNumeric type requirements. This, the result type of such an expression will be another safe type. The actual type of the result of such an expression will depend upon the specific promotion policy template parameter.

Header

`#include <safe/numeric/safe_range.hpp>` [../../include/safe_range.hpp]

Example of use

```
#include <safe/numeric/safe_range.hpp>

void f(){
    using namespace boost::numeric;
    safe_unsigned_range<7, 24> i
    // since the range is included in [0,255], the underlying type of i
    // will be an unsigned char.
    i = 0;  // throws out_of_range exception
    i = 9;  // ok
    i *= 9; // throws out_of_range exception
    i = -1; // throws out_of_range exception
    std::uint8_t j = 4;
    auto k = i + j;

    // the range of i is [7, 24] and the range of j is [0,255]
    // if either or both types are safe types, the result is a safe type
    // determined by promotion policy.  With the default native promotion policy
    // k will be safe<unsigned int>
    static_assert(std::is_same<decltype(k), safe<unsigned int>);
  }
```

6.3. safe_literal<Value>, safe_unsigned_literal<Value>

Description

A safe type which holds a literal value. This is required to be able to initialize other safe type in such a way that exception code is not generated. It is also useful when creating constexpr versions of safe types. It contains one immutable value known at compile time and hence can be used in any constexpr expression.

Template Parameters

Parameter	Type Requirements	Description
`Value`	`Integer`	value used to initialize the literal

Model of

Integer

SafeNumeric

LiteralType [http://en.cppreference.com/w/cpp/concept/LiteralType]

Inherited Valid Expressions

SafeLiteral types are immutable. Hence they only inherit those valid expressions which don't change the value. The excludes assignment, increment, and decrement operators. Other than that, they can be used anywhere a SafeNumeric type can be used.

Header

#include <safe/numeric/safe_literal.hpp> [../../include/safe_range.hpp]

Example of use

```
#include <safe/numeric/safe_literal.hpp>

using namespace boost::numeric;
constexpr safe<int> x = safe_literal<42>{};
```

6.4. Promotion Policies

native

Description

This type contains the functions to return a safe type corresponding to the C++ type resulting for a given arithmetic operation.

Usage of this policy with safe types will produce the exact same arithmetic results that using normal unsafe integer types will. Hence this policy is suitable as a drop-in replacement for these unsafe types. It's main function is to trap incorrect arithmetic results when using C++ for integer arithmetic.

Model of

PromotionPolicy

As an example of how this works consider C++ rules from section 5 of the standard - "usual arithmetic conversions".

```
void int f(int x, int y){
    auto z = x + y; // z will be of type "int"
    return z;
}
```

According to these rules, z will be of type int. Depending on the values of x and y, z may or may not contain the correct arithmetic result of the operation x + y.

```
using safe_int = safe<int, native>;
void int f(safe_int x, safe_int y){
    auto z = x + y; // z will be of type "safe_int"
    return z;
}
```

Header

`#include <boost/safe_numerics/include/native.hpp>` [../../include/native.hpp]

Example of use

The following example illustrates the `native` type being passed as a template parameter for the type `safe<int>`. This example is slightly contrived in that `safe<int>` has `native` as it's default promotion parameter so explicitly using `native` is not necessary.

```
#include <cassert>
#include <boost/safe_numerics/safe_integer.hpp>
#include <boost/safe_numerics/native.hpp>
int main(){
    using namespace boost::numeric;
    // use native promotion policy where C++ standard arithmetic might lead to incorrect results
    using safe_int8 = safe<std::int8_t, native>;
    try{
        safe_int8 x = 127;
        safe_int8 y = 2;
        safe_int8 z;
        // rather than producing and invalid result an exception is thrown
        z = x + y;
        assert(false); // never arrive here
    }
    catch(std::exception & e){
        // which can catch here
        std::cout << e.what() << std::endl;
    }

    // When result is an int, C++ promotion rules guarentee that there will be no incorrect result.
    // In such cases, there is no runtime overhead from using safe types.
    safe_int8 x = 127;
    safe_int8 y = 2;
    safe<int, native> z; // z can now hold the result of the addition of any two 8 bit numbers
    z = x + y; // is guarenteed correct without any runtime overhead or interrupt.

    return 0;
}
```

Notes

See Chapter 5, Expressions, C++ Standard

automatic

Description

This type contains the functions to return a type with sufficient capacity to hold the result of a given arithmetic operation.

Model of

PromotionPolicy

Header

`#include <boost/safe_numerics/automatic.hpp>` [../../include/automatic.hpp]

Example of use

The following example illustrates the `automatic` type being passed as a template parameter for the type `safe<int>`.

```
#include <boost/safe_numerics/safe_integer.hpp>
#include <boost/safe_numerics/automatic.hpp>

int main(){
    using namespace boost::numeric;
    // use automatic promotion policy where C++ standard arithmetic might lead to incorrect results
    using safe_t = safe<std::int8_t, automatic>;

    // In such cases, there is no runtime overhead from using safe types.
    safe_t x = 127;
    safe_t y = 2;
    auto z = x + y; // z is guarenteed correct without any runtime overhead or interrupt.

    return 0;
}
```

cpp<int C, int S, int I, int L, int LL>

Description

This policy is used to promote safe types in arithmetic expressions according the standard rules in the C++ standard. But rather than using the native C++ standard types supported by the compiler, it uses types whose length in number of bits is specified by the template parameters.

This policy is useful for running test program which use C++ portable integer types but which are destined to run on an architecture which is different than the one on which the test program is being built and run. This can happen when developing code for embedded systems. Algorithms developed or borrowed from one architecture but destined for another can be tested on the desk top.

Template Parameters

Parameter	Type	Description
`C`	int	Number of bits in a char
`S`	int	Number of bits in a short
`I`	int	Number of bits in an integer
`L`	int	Number of bits in a long
`LL`	int	Number of bits in a long long

Model of

PromotionPolicy

Header

`#include <boost/safe_numerics/cpp.hpp>` [../../examples/example9.cpp]

Example of Use

Consider the following problem. One is developing software which uses a very small microprocessor and a very limited C compiler. The chip is so small, you can't print anything from the code, log, debug or anything else. One debugs this code by using the "burn" and "crash" method - you burn the chip (download the code) run the code, observe the results, make changes and try again. This is a crude method which is usually the one used. But it can be quite time consuming.

Consider an alternative. Build and compile your code in testable modules. For each module write a test which exercises all the code and makes it work. Finally download your code into the chip and - voilà - working product. This sounds great, but there's one problem. Our target processor - in this case a PIC162550 from Microchip Technology is only an 8 bit CPU. The compiler we use defines INT as 8 bits. This (and a few other problems), make our algorithm testing environment differ from our target environment. We can address this by defining INT as a safe integer with a range of 8 bits. By using a custom promotion policy, we can force the evaluation of C++ expressions test environment to be the same as that in the target environment. Also in our target environment, we can trap any overflows or other errors. So we can write and test our code on our desktop system and download the code to the target knowing that it just has to work. This is a huge time saver and confidence builder. The following code is taken from a real project which has used this method.

```
#include "../include/safe_integer.hpp"
#include "../include/cpp.hpp"

////////////////////////////////////////////////////////
// Stepper Motor Control

// emululate evironment for pic162550
// data widths used by the CCS compiler for pic 16xxx series

using pic16_promotion = boost::numeric::cpp<
    8,  // char
    8,  // short - not used by pic 16xxxx
    8,  // int
    16, // long
    32  // long long
>;

template <typename T> // T is char, int, etc data type
using safe_t = boost::numeric::safe<
    T,
    pic16_promotion,
    boost::numeric::trap_exception // use for compiling and running tests
>;

using int8 = safe_t<std::int8_t>;
using int16 = safe_t<std::int16_t>;
using int32 = safe_t<std::int32_t>;
using uint8 = safe_t<std::uint8_t>;
using uint16 = safe_t<std::uint16_t>;
using uint32 = safe_t<std::uint32_t>;

////////////////////////////////////////////////////////
// Mock defines, functions etc which are in he "real application

...

// return value in steps
/*
Use the formula:
    stopping dist = v **2 / a / 2
*/
```

```
uint16 get_stopping_distance(LEMPARAMETER velocity){
    int32 d;
    d = velocity;
    d *= velocity;
    d /= lem.acceleration;
    d /= 2;
    return d;
}

...
```

Note the usage of the compile time trap policy in order to detect at compile time any possible error conditions. As I write this, this is still being refined. Hopefully this will be available by the time you read this.

6.5. Exception Policies

throw_exception

Description

This exception policy throws a an exception derived from `std::exception` any time some operation would result in an incorrect result. This is the default exception handling policy.

Model of

ExceptionPolicy

Header

`#include <boost/safe_numerics/exception_policies.hpp>` [../../include/exception_policy.hpp]

Example of use

This example is somewhat contrived as throw_exception is the default for safe types. Hence it usually is not necessarily to request it explicitly.

```
#include "../../include/safe_integer.hpp"
#include "../../include/exception_policies" // exception policies
#include "../../include/native"             // native promotion policy

int main(){
    using namespace boost::numeric;
    safe<int, native, throw_exception> i;
    int j = 0;
    i /= j;   // throws exception
}
```

trap_exception

Description

This exception policy will trap at compile time any operation ***COULD*** result in a runtime exception. It can be used in an environment which can tolerate neither arithmetic errors nor runtime overhead. Usage of this policy will almost always require altering one's program to avoid exceptions.

Model of

ExceptionPolicy

Header

```
#include <boost/safe_numerics/exception_policy.hpp>  [../../include/exception_policy.hpp]
```

Example of use

The following

```
#include "../../include/safe_integer.hpp"
#include "../../include/automatic.hpp"
#include "../../include/exception_policies.hpp"

int main(){
    using namespace boost::numeric;
    safe<char, automatic, trap_exception> x, y;
    y = x * x; // compile error here !!!
    auto z = x * x; // compile OK
    return 0;
}
```

ignore_exception

Description

This exception policy can be used to just ignore conditions which generate incorrect arithmetic results and continue processing. Programs using this policy along with the native promotion policy should function as if the library is not even being used.

Model of

ExceptionPolicy

Header

```
#include <boost/safe_numerics/exception_policy.hpp>  [../../include/exception_policy.hpp]
```

Example of use

```
safe<int, native, ignore_exception> st(4);
```

no_exception_support<O, U = O, R =O, D = O>

Description

This exception policy can be used in an environment where one cannot or does not want to use exceptions.

Parameters are pointers to static functions which are invoked for each kind of error encountered in the library. The function signature of these functions are `void function(const char * message)` where `message` is the address of a literal string with information regarding the error.

Template Parameters

Function objects to be invoked are specified for each error condition are specified via template parameters.

Parameter	Default	Type Requirements	Description
O		void (*O)(const char *)	Function to call on overflow error
O	O	void (*U)(const char *)	Function to call on underflow error
O	O	void (*R)(const char *)	Function to call on range error
O	O	void (*D)(const char *)	Function to call on domain error

Model of

ExceptionPolicy

Header

```
#include <boost/safe_numerics/exception_policy.hpp>  [../../include/exception_policy.hpp]
```

Example of use

[A code fragment involving the type.]

```
void overflow(const char * msg);
void underflow(const char * msg);
void range_error(const char * msg);
void domain_error(const char * msg);

using ep = ignore_exception<
    overflow,
    underflow,
    range_error,
    domain_error
>;

safe<int, native, ep> st(4);
```

7. Exception Safety

All operations in this library are exception safe and meet the strong guarantee.

8. Library Implementation

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. The 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. They are not dependent upon anything in the Safe Numerics library. Rather it's the other way around. The Safe Numerics library depends upon these components. It is thought that they may be useful outside of the context of the Safe Numerics library. For all these reasons, they are documented here.

8.1. exception_type

Description

This `enum` holds the information regarding a failed operation.

Notation

Symbol	Description
`C`	checked_result<R>
`c`	an instance of checked_result<R>
`t`	an instance of checked_result<T> for some type T not necessarily the same as R
`r`	An object of type R
`EP`	ExceptionPolicy type
`e`	`enum` class exception_type value
`msg`	`const char *` - pointer to static string with error message

Valid Expressions

In addition to the expressions defined in Random Access Container [http://www.sgi.com/tech/stl/RandomAccessContainer.html] and Back Insertion Sequence [http://www.sgi.com/tech/stl/BackInsertionSequence.html] the following expressions must be valid for any object `v` of type `vector<T>` .

Expression	Semantics

`exception_type::no_exception`
`exception_type::everflow_error`
`exception_type::underflow_error`
`exception_type::range_error`
`exception_type::domain_error`
`exception_type::uninitialized`
	enum values which describe various type of error conditions

`dispatch<EP>(e, msg);`	invoke the exception handling policy for a particular exception_type

dispatch<EP>(const exception_type & e, const char * msg)

This function is used to invoke the exception handling policy for a particular exception_type.

Synopsis

```
template<class EP>
constexpr void
dispatch<EP>(const exception_type & e, const char * msg);
```

Example of use

```
#include <vector>

vector<int> V;
V.insert(V.begin(), 3);
assert(V.size() == 1 && V.capacity() >= 1 && V[0] == 3);
```

See Also

ExceptionPolicy

8.2. checked_result<typename R>

Description

checked_result is a wrapper class designed to hold result of some operation. It can hold either the result of the operation or information on why the operation failed to produce a valid result. Note that this type is an internal feature of the library and shouldn't be exposed to library users because it has some unsafe behavior.

Template Parameters

The sole template parameter is the return type of some operation.

Parameter	Default	Type Requirements	Description
`R`			The value type.

Notation

Symbol	Description
`C`	checked_result<R>
`c`	an instance of checked_result<R>
`t`	an instance of checked_result<T> for some type T not necessarily the same as R
`r`	An object of type R
`EP`	ExceptionPolicy

Valid Expressions

All expressions are constexpr.

Expression	Return Type	Semantics
`checked_result(r)`	`checked_result<R>`	constructor with valid instance of R
`checked_result(e, msg)`	`checked_result<R>`	constructor with error information

Expression	Return Type	Semantics
`static_cast<R>(c)`	R	extract wrapped value - asserts if not possible
`static_cast<const char *>(c)`	R	extract wrapped value - asserts if not possible

`c < t`
`c >= t`
`c > t`
`c <= t`
`c == t`
`c != t`
	boost::logic::tribool	compare the wrapped values of two checked_result instances. If either one contains an invalid value, return boost::logic::tribool::indeterminant.

`c.no_exception()`	bool	true if checked_result contains a valid result
`c.exception()`	bool	true if checked_result contains an error condition
`dispatch<EP>(c)`	void	invoke exception in accordance exception_type value

Header

```
#include "checked_result.hpp"
```

Example of use

```
#include "checked_result.hpp"

template<class R>
checked_result<R>
constexpr modulus(
    const R & t,
    const R & u
) {
    if(0 == u)
        return checked_result<R>(
            exception_type::domain_error,
            "denominator is zero"
        );

    return t % u;
}
```

See Also

ExceptionPolicy

8.3. Checked Integer Arithmetic

Synopsis

```
// safe casting on primitive types
template<class R, class T>
```

```
constexpr checked_result<R>
checked::cast(const T & t);

// safe addition on primitive types
template<class R, class T, class U>
constexpr checked_result<R>
checked::add(const T & t, const U & u);

// safe subtraction on primitive types
template<class R, class T, class U>
constexpr checked_result<R>
checked::subtract(const T & t, const U & u);

// safe multiplication on primitive types
template<class R, class T, class U>
constexpr checked_result<R>
checked::multiply(const T & t, const U & u);

// safe division on unsafe types
template<class R, class T, class U>
constexpr checked_result<R>
checked::divide(const T & t, const U & u);

template<class R, class T, class U>
constexpr checked_result<R>
checked::divide_automatic(const T & t, const U & u);

// safe modulus on unsafe types
template<class R, class T, class U>
constexpr checked_result<R>
checked::modulus(const T & t, const U & u);

// left shift
template<class R, class T, class U>
constexpr checked_result<R>
checked::left_shift(const T & t, const U & u);

// right shift
template<class R, class T, class U>
constexpr checked_result<R>
checked::right_shift(const T & t, const U & u);

// bitwise operations
template<class R, class T, class U>
constexpr checked_result<R>
checked::bitwise_or(const T & t, const U & u);

template<class R, class T, class U>
constexpr checked_result<R>
checked::bitwise_and(const T & t, const U & u);

template<class R, class T, class U>
constexpr checked_result<R>
checked::bitwise_xor(const T & t, const U & u);
```

Description

Perform binary operations on arithmetic types. Return either a valid result or an error code. Under no circumstances should an incorrect result be returned.

Type requirements

All template parameters of the functions must model Integer type requirements.

Complexity

Each function performs one and only one arithmetic operation

Header

```
#include "checked.hpp"
```

Example of use

[A code fragment that illustrates how to use the function.]

```
#include "checked.hpp"

checked_result<result_base_type> r = checked::multiply<int>(24, 42);
```

Notes

Some compilers have command line switches (e.g. -ftrapv) which enable special behavior such erroneous integer operations are detected at run time. The library has been implemented in such a way that these facilities are not used. It's possible they might be helpful in particular environment. These could be be exploited by re-implementing some functions in this library.

See Also

checked_result<typename R>

8.4. interval<typename R>

Description

A closed arithmetic interval represented by a pair of elements of type R.

Template Parameters

R must model the type requirements Numeric

Notation

Symbol	Description
`I`	An interval type
`i, j`	An interval
`R, T`	Numeric types which can be used to make an interval
`l, u`	lower and upper Numeric limits of an interval

Symbol	Description
C	checked_result<interval<R>>
os	std::basic_ostream<Char, CharT>

Associated Types

checked_result	holds either the result of an operation or information as to why it failed

Valid Expressions

Note that all expressions are constexpr .

Expression	Return Type	Semantics
interval<R>(l, u)	interval<R>	construct a new interval from a pair of limits
interval<R>(p)	interval<R>	construct a new interval from a pair of limits
interval<R>(i)	interval<R>	copy constructor
i.includes(j)	bool	return true if interval i includes interval j
i.includes(t)	bool	return true if interval i includes value t
add<R>(i, j)	C	add two intervals and return the result
subtract<R>(i, j)	C	subtract two intervals and return the result
multiply<R>(i, j)	C	multiply two intervals and return the result
divide_nz<R>(i, j)	C	divide one interval by another excluding the value zero and return the result
divide<R>(i, j)	C	divide one interval by another and return the result
modulus_nz<R>(i, j)	C	calculate modulus of one interval by another excluding the value zero and return the result
modulus<R>(i, j)	C	calculate modulus of one interval by another and return the result
left_shift<R>(i, j)	C	calculate the range that would result from shifting one interval by another
right_shift<R>(i, j)	C	calculate the range that would result from shifting one interval by another
t < u	boost::logic::tribool	true if every element in t is less than every element in u
t > u	boost::logic::tribool	true if every element in t is greater than every element in u

Expression	Return Type	Semantics
`t <= u`	`boost::logic::tribool`	true if every element in t is less than or equal to every element in u
`t >= u`	`boost::logic::tribool`	true if every element in t is greater than or equal to every element in u
`t == u`	`bool`	true if limits are equal
`t != u`	`bool`	true if limits are not equal
`os << i`	`os &`	print interval to output stream

Header

```
#include "interval.hpp"
```

Example of use

```
#include <iostream>
#include <cstdint>
#include <cassert>
#include <boost/numeric/interval.hpp>

int main(){
    std::cout << "test1" << std::endl;
    interval<std::int16_t> x = {-64, 63};
    std::cout << "x = " << x << std::endl;
    interval<std::int16_t> y(-128, 126);
    std::cout << "y = " << y << std::endl;
    assert(static_cast<interval<std::int16_t>>(add<std::int16_t>(x,x)) == y);
    std::cout << "x + x =" << add<std::int16_t>(x, x) << std::endl;
    std::cout << "x - x = " << subtract<std::int16_t>(x, x) << std::endl;
    return 0;
}
```

9. Performance Tests

Our goal creating facilities which make it possible write programs known to be correct. But we also want programmers to actually use the facilities we provide here. This won't happen if using these facilities impacts performance to a significant degree. Although we've take precautions to avoid doing this, the only real way to know is to create and run some tests.

To Do

10. Rationale and FAQ

10.1. 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 cases in the section Tutorials and Motivating Examples.

10.2. 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. See drop-in replacement.

10.3. Why is Boost.Convert not used?

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

10.4. 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

10.5. 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> et. al. and eventually 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 it's name.

10.6. 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], Software Engineering Institute - Carnegie Mellon University [https://www.cert.org]

10.7. 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.

10.8. 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.

10.9. According to C/C++ standards, unsigned integers cannot overflow - they are modular integers which "warp 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 modular type, the plain reading of an arithmetic expression conveys the idea that all operands are plain 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 which is expected to provide correct integer results. 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 2008]..

11. Pending Issues

The library is under development. There are a number of issues still pending.

- 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.

12. Change Log

This is the third version.

13. Bibliograph

Zack Coker. Samir Hasan. Jeffrey Overbey. Munawar Hafiz. Christian Kästner. *Integers In C: An Open Invitation To Security Attacks? [https://www.cs.cmu.edu/~ckaestne/pdf/csse14-01.pdf]* *[http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]* . JTC1/SC22/WG21 - The C++ Standards Committee - ISOCPP [http://www.open-std.org/jtc1/sc22/wg21/] . January 15, 2012. Crowl

Lawrence Crowl. *C++ Binary Fixed-Point Arithmetic [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3352.html]* *[http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]* . JTC1/SC22/WG21 - The C++ Standards Committee - ISOCPP [http://www.open-std.org/jtc1/sc22/wg21/] . January 15, 2012. Crowl

Lawrence Crowl. Thorsten Ottosen. *Proposal to add Contract Programming to C++ [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1962.html]* *[http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]* . WG21/N1962 and J16/06-0032 - The C++ Standards Committee - ISOCPP [http://www.open-std.org/jtc1/sc22/wg21/] . February 25, 2006. Crowl & Ottosen

Will Dietz. Peng Li. John Regehr. Vikram Adve. *Understanding Integer Overflow in C/C++ [http://www.cs.utah.edu/~regehr/papers/overflow12.pdf]* . Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland [http://dl.acm.org/citation.cfm?id=2337223&picked=prox] . June 2012.

J. Daniel Garcia. *C++ language support for contract programming [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1962.html]* *[http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]* . WG21/N4293 - The C++ Standards Committee - ISOCPP [http://www.open-std.org/jtc1/sc22/wg21/] . December 23, 2014. Garcia

Omer Katz. *SafeInt code proposal [http://boost.2283326.n4.nabble.com/SafeInt-code-proposal-td2663669.html]* *[http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]* . Boost Developer's List [https://groups.google.com/a/isocpp.org/forum/?fromgroups#!forum/std-proposals] . Katz

David LeBlanc. *Integer Handling with the C++ SafeInt Class* *[https://msdn.microsoft.com/en-us/library/ms972705.aspx]* . Microsoft Developer Network [https://www.cert.org] . January 7, 2004. LeBlanc

David LeBlanc. *SafeInt [https://safeint.codeplex.com]* . CodePlex [https://www.cert.org] . Dec 3, 2014. LeBlanc

Jacques-Louis Lions. *Ariane 501 Inquiry Board report* *[https://en.wikisource.org/wiki/Ariane_501_Inquiry_Board_report]* . Wikisource [https://en.wikisource.org/wiki/Main_Page] . July 19, 1996. Lions

Jad Mouawad. *F.A.A Orders Fix for Possible Power Loss in Boeing 787 [http://www.nytimes.com/2015/05/01/business/faa-orders-fix-for-possible-power-loss-in-boeing-787.html?_r=0]* *[http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]* . New York Times. April 30, 2015.

Daniel Plakosh. *Safe Integer Operations [https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/coding/312-BSI.html]* . U.S. Department of Homeland Security [https://buildsecurityin.us-cert.gov] . May 10, 2013. Plakosh

[Seacord] Robert C. Seacord. *Secure Coding in C and C++ [http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]* . 2nd Edition. Addison-Wesley Professional. April 12, 2013. 978-0321822130. Seacord

Robert C. Seacord. *INT30-C. Ensure that operations on unsigned integers do not wrap [https://www.securecoding.cert.org/confluence/display/seccode/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow?showComments=false]* . Software Engineering Institute, Carnegie Mellon University [https://www.cert.org] . August 17, 2014. INT30-C

Robert C. Seacord. *INT32-C. Ensure that operations on signed integers do not result in overflow [https://www.securecoding.cert.org/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow]* . Software Engineering Institute, Carnegie Mellon University [https://www.cert.org] . August 17, 2014. INT32-C

Forum Posts. *C++ Binary Fixed-Point Arithmetic [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3352.html] [http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]* . ISO C++ Standard Future Proposals [https://groups.google.com/a/isocpp.org/forum/?fromgroups#!forum/std-proposals] . Forum

David Keaton, Thomas Plum, Robert C. Seacord, David Svoboda, Alex Volkovitsky, and Timothy Wilson. *As-if Infinitely Ranged Integer Model [http://resources.sei.cmu.edu/asset_files/TechnicalNote/2009_004_001_15074.pdf] [http://www.cert.org/secure-coding/publications/books/secure-coding-c-c-second-edition.cfm?]* . Software Engineering Institute [http://www.sei.cmu.edu] . CMU/SEI-2009-TN-023.