
Chapter 1. Safe Numerics

Robert Ramey

Copyright © 2012 Robert Ramey

[Subject to Boost Software License](#)

Table of Contents

1. Introduction	4
1.1. Problem	4
1.2. Solution	5
1.3. How It Works	5
1.4. Additional Features	6
1.5. Requirements	6
1.6. Scope	7
2. Tutorial and Motivating Examples	7
2.1. Arithmetic Expressions Can Yield Incorrect Results	7
2.2. Arithmetic Operations Can Overflow Silently	8
2.3. Implicit Conversions Can Lead to Erroneous Results	9
2.4. Mixing Data Types Can Create Subtle Errors	10
2.5. Array Index Value Can Exceed Array Limits	12
2.6. Checking of Input Values Can Be Easily Overlooked	13
2.7. Cannot Recover From Arithmetic Errors	14
2.8. Programming by Contract is Too Slow	15
3. Case Studies	17
3.1. Eliminating Runtime Penalty	17
Using <code>safe_range</code> and <code>safe_literal</code>	17
Using Automatic Type Promotion	19
Mixing Approaches	21
3.2. Composition with Other Libraries	22
4. Background	24
5. Type Requirements	24
5.1. <code>Numeric<T></code>	24
Description	24
Notation	24
Associated Types	25
Valid Expressions	25
Models	26
Header	26
5.2. <code>Integer<T></code>	26
Description	26
Refinement of	27
Notation	27
Valid Expressions	27
Models	28
Header	28
5.3. <code>SafeNumeric<T></code>	28
Description	28
Refinement of	28
Notation	28

Valid Expressions	28
Complexity Guarantees	30
Invariants	30
Models	30
Header	30
5.4. PromotionPolicy<PP>	30
Description	30
Notation	30
Valid Expressions	30
Models	31
Header	32
5.5. ExceptionPolicy<EP>	32
Description	32
Notation	32
Valid Expressions	32
dispatch<EP>(const safe_numerics_error & e, const char * msg)	33
Models	33
Header	33
6. Types	34
6.1. safe<T, PP, EP>	34
Description	34
Notation	34
Associated Types	34
Template Parameters	34
Model of	34
Valid Expressions	34
Examples of use	35
Header	37
6.2. safe_signed_range<MIN, MAX, PP, EP> and safe_unsigned_range<MIN, MAX, PP, EP>	37
Description	37
Notation	37
Associated Types	38
Template Parameters	38
Model of	38
Valid Expressions	38
Example of use	38
Header	39
6.3. safe_signed_literal<Value, PP, EP> and safe_unsigned_literal<Value, PP, EP>	39
Description	39
Associated Types	39
Template Parameters	39
Model of	39
Inherited Valid Expressions	40
New Valid Expression	40
Example of use	40
Header	41
6.4. exception	41
Description	41
enum class safe_numerics_error	41
enum class safe_numerics_actions	42
See Also	42
Header	42
6.5. exception_policy<AE, IDB, UB, UV>	42
Description	42

Template Parameters	42
Model of	43
Inherited Valid Expressions	43
Function Objects	43
Policies Provided by the library	43
Header	44
6.6. Promotion Policies	44
native	44
automatic	46
cpp<int C, int S, int I, int L, int LL>	47
7. Exception Safety	49
8. Library Implementation	49
8.1. checked_result<R>	50
Description	50
Template Parameters	50
Notation	50
Valid Expressions	50
Example of use	51
See Also	51
Header	51
8.2. Checked Arithmetic	51
Synopsis	51
Description	52
Type requirements	53
Complexity	53
Example of use	53
Notes	53
See Also	53
Header	53
8.3. interval<typename R>	53
Description	53
Template Parameters	53
Notation	53
Associated Types	54
Valid Expressions	54
Example of use	55
Header	55
8.4. safe_compare<T, U>	51
Synopsis	55
Description	56
Type requirements	56
Complexity	56
Example of use	56
Header	56
9. Performance Tests	56
10. Rationale and FAQ	57
11. Pending Issues	60
12. Acknowledgements	60
13. Change Log	61
14. Bibliography	61

1. Introduction

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

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

1.1. Problem

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

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

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

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

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

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

This example addresses only the problem of undefined/erroneous behavior related to overflow of the addition operation as applied to the type `int`. Similar problems occur with other built-in integer types such as `unsigned`, `long`, etc. And it also applies to other operations such as subtraction, multiplication etc. . C/C++ often automatically and silently converts some integer types to others in the course of implementing binary operations. Sometimes such conversions can silently change arithmetic values which inject errors. The C/C++ standards designate some behavior such as right shifting a negative number as "implementation defined behavior". These days machines usually do what the programmer expects - but such behavior is not guaranteed. Relying on such behavior will create a program which cannot be guaranteed to be portable. And then there is undefined behavior. In this case,

compiler writer is under no obligation to do anything in particular. Sometimes this will break the program. At the very least, the program is rendered non-portable. Finally there is the case of behavior that is arithmetically wrong to begin with - for example divide by zero. Some runtime environments will just terminate the program, others may throw some sort of exception. In any case, the execution has failed in a manner from which there is no recovery.

All of the above conditions prevent obstacles to creation of a program which will never fail. The Safe Numerics Library addresses all of these conditions, at least as far as integer operations are concerned.

Since the problems and their solution are similar, We'll confine the current discussion to just the one example shown above.

1.2. Solution

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

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

Note

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

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

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

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

1.3. How It Works

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

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

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

1.4. Additional Features

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

- Select or define an exception policy class to specify handling of exceptions.
 - Throw exception on runtime, trap at compile time if possible.
 - Trap at compiler time all operations which could possibly fail at runtime.
 - Specify custom functions which should be called in case errors are detected at runtime.
- Select or define a promotion policy class to alter the C/C++ type promotion rules. This can be used to
 - Use C/C++ native type promotion rules so that, except for throwing/trapping of exceptions on operations resulting in incorrect arithmetic behavior, programs will operate identically when using/not using safe types. This might be used if safe types are only enabled during debug and testing.
 - Replace C/C++ native promotion rules with ones which are arithmetically equivalent but minimize the need for runtime checking of arithmetic results. Such a policy will effectively change the semantics of a C++ program. It's not really C++ any more. The program cannot be expected to function the same as when normal integer types are used.
 - Replace C/C++ native promotion rules with ones which emulate other machine architectures. This is designed to permit the testing of C/C++ code destined to be run on another machine on one's development platform. Such a situation often occurs while developing code for embedded systems.
- Enforce other program requirements using bounded integer types. The library includes the types for ranges and literals. Operations which result in operations which violate these requirements will be trapped at either compile or runtime and not silently return invalid values. 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
- Integer
- Core

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

1.6. Scope

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

2. Tutorial and Motivating Examples

2.1. Arithmetic Expressions Can Yield Incorrect Results

When some operation on signed integer types results in a result which exceeds the capacity of a data variable to hold it, the result is undefined. In the case of unsigned integer types a similar situation results in a value wrap as per modulo arithmetic. In either case the result is different than in integer number arithmetic in the mathematical sense. This is called "overflow". Since word size can differ between machines, code which produces mathematically 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 `int` type with `safe<int>` 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;
    try{
        int x = INT_MAX;
        int y = 2;
        int z;
        // this produces an invalid result !
        z = x + y;
        std::cout << z << " != " << x << " + " << y << 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;
        safe<int> y = 2;
        safe<int> z;
        // rather than producing an invalid result an exception is thrown
        z = x + y;
    }
    catch(std::exception & e){
        // which we can catch here
```

```
        std::cout << "error detected:" << e.what() << std::endl;
    }
    return 0;
}
```

```
example 1:undetected erroneous expression evaluation
Not using safe numerics
-2147483647 != 2147483647 + 2
error NOT detected!
Using safe numerics
error detected:addition result too large: positive overflow error
```

2.2. Arithmetic Operations Can Overflow Silently

A variation of the above is when a value is incremented/decremented beyond its 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;
}
```

```
example 2:undetected overflow in data type
Not using safe numerics
```



```
-2147483648 != 2147483647 + 1
error NOT detected!
Using safe numerics
addition result too large
error detected!
```

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 Can Lead to Erroneous Results

At CPPCon 2016 Jon Kalb gave a very entertaining (and disturbing) [lightning talk](#) related to C++ expressions.

The talk included a very, very simple example similar to the following:

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

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

int main(){
    std::cout << "example 3: ";
    std::cout << "implicit conversions change data values" << std::endl;
    std::cout << "Not using safe numerics" << std::endl;

    // problem: implicit conversions change data values
    try{
        signed int    a{-1};
        unsigned int  b{1};
        std::cout << "a is " << a << " b is " << b << '\n';
        if(a < b){
            std::cout << "a is less than b\n";
        }
        else{
            std::cout << "b is less than a\n";
        }
        std::cout << "error NOT detected!" << std::endl;
    }
    catch(std::exception){
        // never arrive here - just produce the wrong answer!
        std::cout << "error detected!" << std::endl;
        return 1;
    }

    // solution: replace int with safe<int> and unsigned int with safe<unsigned int>
    std::cout << "Using safe numerics" << std::endl;
    try{
        using namespace boost::numeric;
        safe<signed int>    a{-1};
        safe<unsigned int> b{1};
        std::cout << "a is " << a << " b is " << b << '\n';
        if(a < b){
            std::cout << "a is less than b\n";
        }
        else{
            std::cout << "b is less than a\n";
        }
        std::cout << "error NOT detected!" << std::endl;
    }
```

```
        return 1;
    }
    catch(std::exception & e){
        // never arrive here - just produce the correct answer!
        std::cout << e.what() << std::endl;
        std::cout << "error detected!" << std::endl;
    }
    return 0;
}
```

```
example 3: implicit conversions change data values
Not using safe numerics
a is -1 b is 1
b is less than a
error NOT detected!
Using safe numerics
a is -1 b is 1
converted negative value to unsigned: domain error
error detected!
```

A normal person reads the above code and has to be dumbfounded by this. The code doesn't do what the text - according to the rules of algebra - says it does. But C++ doesn't follow the rules of algebra - it has its own rules. There is generally no compile time error. You can get a compile time warning if you set some specific compile time switches. The explanation lies in reviewing how C++ reconciles binary expressions (`a < b` is an expression here) where operands are different types. In processing this expression, the compiler:

- Determines the "best" common type for the two operands. In this case, application of the rules in the C++ standard dictate that this type will be an unsigned int.
- Converts each operand to this common type. The signed value of -1 is converted to an unsigned value with the same bit-wise contents, 0xFFFFFFFF, on a machine with 32 bit integers. This corresponds to a decimal value of 4294967295.
- Performs the calculation - in this case it's <, the "less than" operation. Since 1 is less than 4294967295 the program prints "b is less than a".

In order for a programmer to detect and understand this error he should be pretty familiar with the implicit conversion rules of the C++ standard. These are available in a copy of the standard and also in the canonical reference book [The C++ Programming Language](#) (both are over 1200 pages long!). Even experienced programmers won't spot this issue and know to take precautions to avoid it. And this is a relatively easy one to spot. In the more general case this will use integers which don't correspond to easily recognizable numbers and/or will be buried as a part of some more complex expression.

This example generated a good amount of web traffic along with everyone's pet suggestions. See for example [a blog post with everyone's favorite "solution"](#). All the proposed "solutions" have disadvantages and attempts to agree on how handle this are ultimately fruitless in spite of, or maybe because of, the [emotional content](#). Our solution is by far the simplest: just use the safe numerics library as shown in the example above.

Note that in this particular case, there is absolutely no extra overhead in using the safe integer library. Code generated will either equal or exceed the efficiency of using primitive integer types.

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"

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 4: ";
    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
        cout << "error NOT detected!" << endl;;
    }
    catch(const std::exception & e){
        // never arrive here
        cout << "error detected:" << e.what() << 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
        cout << "error NOT detected!" << endl;;
    }
    catch(const std::exception & e){
        cout << "error detected:" << e.what() << endl;;
    }
    return 0;
}
```

Here is the output of the above program:

```
example 4: mixing types produces surprising results
Not using safe numerics
10000
4294957296
error NOT detected!
Using safe numerics
10000
error detected!converted negative value to unsigned: domain error
```

This solution is simple, just replace instances of `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_integer_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 << "error detected:" << e.what() << std::endl;
    }
    return 0;
}
```

```
example 5: array index values can exceed array bounds
Not using safe numerics
error NOT detected!
Using safe numerics
error detected:Value out of range for this safe type: domain error
```

Note the usage of `static_cast` in order to convert an instance of a safe type, `i_index`, to a built-in integer type in order to index into an array with the expression `i_array[]`. This is necessary because conversion from a safe type to its underlying built-in type is marked with `explicit`. This is intentional. Although it hampers readability and subverts the design goal of making safe types "drop-in" replacements for their underlying built-in types, it has been deemed that permitted implicit (and therefore silent) conversions to unsafe times will inhibit the inadvertent loss of "safeness". If one uses a safe type where a built-in type is expected, a syntax error will result - thus requiring the programmer to actively decide how he might want to handle this. He can always use

an explicit `safe_cast` to convert the safe type to the underlying type. But in some cases he might want to change the target type to a safe type there by propagating the "safeness" to other types in the program. For more on this topic see [Rationale](#).

Collections like standard arrays and 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 ranges 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 1231287389217389217893");

    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: assign 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 << "error detected:" << e.what() << std::endl;
        }
    }
    return 0;
}
```

```
example 6: checking of externally produced value can be overlooked
Not using safe numerics
2147483647 0
error NOT detected!
```

```
Using safe numerics
error detected:error in file input: domain error
```

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. Cannot Recover From Arithmetic Errors

If a divide by zero error occurs in a program, it's detected by hardware. The way this manifests itself to the program can and will depend upon

- data type - int, float, etc
- setting of compile time command line switches
- invocation of some configuration functions which convert these hardware events into C++ exceptions

It's not all that clear how one would detect and recover from a divide by zero error in a simple portable way. Usually, users just ignore the issue which usually results in immediate program termination when this situation occurs.

This library will detect divide by zero errors before the operation is invoked. Any errors of this nature are handled according to the `ErrorPolicy` selected by the library user.

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

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

int main(int argc, const char * argv[]){
    // problem: cannot recover from arithmetic errors
    std::cout << "example 7: ";
    std::cout << "cannot recover from arithmetic errors" << std::endl;
    std::cout << "Not using safe numerics" << std::endl;

    try{
        int x = 1;
        int y = 0;
        // can't do this as it will crash the program with no
        // opportunity for recovery - comment out for example
        //std::cout << x / y;
        std::cout << "error NOT detectable!" << 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 = 1;
        safe<int> y = 0;
        std::cout << x / y;
        std::cout << " error NOT detected!" << std::endl;
    }
    catch(std::exception & e){
        std::cout << "error detected:" << e.what() << std::endl;
    }
}
```

```
    return 0;
}
```

```
example 7: cannot recover from arithmetic errors
Not using safe numerics
error NOT detectable!
Using safe numerics
error detected:divide by zero: domain error
```

2.8. Programming by Contract is Too Slow

Programming by Contract is a highly regarded technique. There has been much written about it and 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 guaranteed parameter requirements with little or no runtime overhead. Consider the following example:

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

#include "../include/safe_integer_range.hpp"

// NOT using safe numerics - enforce program contract explicitly
// return total number of minutes
unsigned int contract_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 convenient 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>;
using minutes_total_t = boost::numeric::safe_unsigned_range<0, 59>;

// return total number of minutes
// type returned is safe_unsigned_range<0, 24*60 - 1>
auto convert(const hours_t & hours, const minutes_t & minutes) {
    // no need to test pre-conditions
    // input parameters are guaranteed to hold legitimate values
```

```

    // no need to test post-conditions
    // return value guaranteed to hold result
    return hours * 60 + minutes;
}

unsigned int test1(unsigned int hours, unsigned int minutes){
    // problem: checking of externally produced value can be expensive
    // invalid parameters - detected - but at a heavy cost
    return contract_convert(hours, minutes);
}

auto test2(unsigned int hours, unsigned int minutes){
    // solution: use safe numerics
    // safe types can be implicitly constructed base types
    // construction guarantees correctness
    // return value is known to fit in unsigned int
    return convert(hours, minutes);
}

auto test3(unsigned int hours, unsigned int minutes){
    // actually we don't even need the convert function any more
    return hours_t(hours) * 60 + minutes_t(minutes);
}

int main(int argc, const char * argv[]){
    std::cout << "example 7: ";
    std::cout << "enforce contracts with zero runtime cost" << std::endl;

    unsigned int total_minutes;

    try {
        total_minutes = test3(17, 83);
        std::cout << "total minutes = " << total_minutes << std::endl;
    }
    catch(std::exception e){
        std::cout << "parameter error detected" << std::endl;
    }

    try {
        total_minutes = test3(17, 10);
        std::cout << "total minutes = " << total_minutes << std::endl;
    }
    catch(std::exception e){
        // should never arrive here
        std::cout << "parameter error erroneously detected" << std::endl;
        return 1;
    }
    return 0;
}

```

```

example 8:
enforce contracts with zero runtime cost
parameter error detected

```

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 the 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. Case Studies

3.1. Eliminating Runtime Penalty

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

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

```
#include <iostream>

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

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

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

Now, any expression which *might* fail at runtime is flagged with a compile time error. There is no longer any need for `try/catch` blocks. Since this program does not compile, the **library absolutely guarantees that no arithmetic expression will yield incorrect results**. Furthermore, it is **absolutely guaranteed that no exception will ever be thrown**. This is our original goal.

Now all we need to do is make the program work. There are a couple of ways to achieve this.

Using `safe_range` and `safe_literal`

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

```
#include <iostream>

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

using namespace boost::numeric;

// create a type for holding small integers in a specific range
```

```

using safe_t = safe_signed_range<
    -24,
    82,
    native,           // C++ type promotion rules work OK for this example
    loose_trap_policy // catch problems at compile time
>;

// create a type to hold one specific value
template<int I>
using const_safe_t = safe_signed_literal<I, native, loose_trap_policy>;

// We "know" that C++ type promotion rules will work such that
// addition will never overflow. If we change the program to break this,
// the usage of the loose_trap_policy promotion policy will prevent compilation.
int main(int argc, const char * argv[]){
    std::cout << "example 83:\n";

    constexpr const const_safe_t<10> x;
    std::cout << "x = " << safe_format(x) << std::endl;
    constexpr const const_safe_t<67> y;
    std::cout << "y = " << safe_format(y) << std::endl;

    constexpr const safe_t z = x + y;
    std::cout << "z = " << safe_format(z) << std::endl;

    const auto za = x + y;
    std::cout << "za = " << safe_format(za) << std::endl;
    return 0;
}

```

- `safe_signed_range` defines a type which is limited to the indicated range. Out of range assignments will be detected at compile time if possible (as in this case) or at run time if necessary.
- A safe range could be defined with the same minimum and maximum value effectively restricting the type to holding one specific value. This is what `safe_signed_literal` does.
- Defining constants with `safe_signed_literal` enables the library to correctly anticipate the correct range of the results of arithmetic expressions.
- The usage of `loose_trap_policy` will mean that any assignment to `z` which could be outside its legal range will result in a compile time error.
- All safe integer operations are implemented as constant expressions. The usage of `constexpr` will guarantee that `z` will be available at compile time for any subsequent use.
- So if this program compiles, it's guaranteed to return a valid result.

This program produces the following run time output.

```

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

```

Take note of the various variable types:

- `x` and `y` are safe types with fixed ranges which encompass one single value. They can hold only that value which they have been assigned at compile time.

- `z` is the sum of `x` and `y`. The type of `z` is defined so that It can hold only values in the closed range `-24,82`. We can assign the sum of `x` and `y` because it is in the range that `z` is guaranteed to hold. If the sum or not known at compile time, and it cannot be guaranteed to fall in the range of `z`, we would get a compile time error due to the fact we are using the `loose_trap_policy` exception policy.
- `za` is sum of `x` and `y` both of which are known a compile time. The type of the result is also known at compile which can be seen from the fact that it is an integer value (since it is the result of native C++ promotion rules) in the range `77,77`. That is, it is guaranteed to be equal to `77`.

All this information regarding the values of variables has been determined at compile time. There is no runtime overhead. The usage of safe types does not alter the calculations or results in anyway. So `safe_t` and `const_safe_t` could be redefined to `int` and `const int` respectively and the program would operate identically. We could compile the program for another machine - as is common when building embedded systems and know (assuming the target machine architecture was the same as our native one) that no erroneous results would ever be produced.

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 size of the signed operand is larger than the size of the signed operand, the type of the result will be signed. Otherwise, the type of the result will be unsigned.
- Convert the type each operand to the type of the result, expanding the size as necessary.
- Perform the operation the two resultant operands.

So the type of the result of some binary operation may be different than the types of either or both of the original operands.

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

The complete signature for a safe integer type is:

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

The promotion rules implemented in the default `native` type promotion policy are consistent with those of standard C++

Up until now, we've focused on detecting when an arithmetic error occurs and invoking an exception or other kind of error handler.

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

- for any C++ numeric type, we know from `std::numeric_limits` what the maximum and minimum values that a variable can be - this defines a closed interval.
- For any binary operation on these types, we can calculate the interval of the result at compile time.

- From this interval we can select a new type which can be guaranteed to hold the result and use this for the calculation. This is more or less equivalent to the following code:

```
int x, y;
int z = x + y           // could overflow
// so replace with the following:
int x, y;
long z = (long)x + (long)y; // can never overflow
```

One could do this by editing his code manually as above, but such a task would be tedious, error prone, non-portable and leave the resulting code hard to read and verify. Using the `automatic` type promotion policy will achieve the equivalent result without these problems.

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

If the interval of the result cannot be guaranteed to fit 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. So while our `automatic` type promotion policy might eliminate exceptions in our example above, it wouldn't be guaranteed to eliminate them for all programs.

Using the `loose_trap_policy` exception policy will produce a compile time error anytime it's possible for an error to occur.

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

```
#include <iostream>

#include "../include/safe_integer.hpp"
#include "../include/exception_policies.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::loose_trap_policy
>;

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

- the `automatic` type promotion policy has rendered the result of the sum of two integers as a `safe<long>` type.
- our program compiles without error - even when using the `loose_trap_policy` exception policy
- We do not need to use the `try/catch` idiom to handle arithmetic errors - we will have no exceptions.
- We only needed to change two lines of code to achieve our goal of guaranteed program correctness with no runtime penalty.

The above program produces the following output:

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

The output uses a custom output manipulator for safe types to display the underlying type and its range as well as current value. Note that if any time in the future we were to change `safe<int>` to `safe<long long>` the program could now overflow. But since we're using `loose_trap_policy` the modified program would fail to compile. At this point we'd have to alter our yet program again to eliminate run time penalty or set aside our goal of zero run time overhead and change the exception policy to `default_exception_policy`.

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_integer.hpp"
#include "../include/safe_integer_range.hpp"
#include "../include/automatic.hpp"
#include "../include/exception.hpp"

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

using namespace boost::numeric;

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

// define variables used for input
using input_safe_t = safe_signed_range<
    -24,
    82,
    automatic, // we don't need automatic in this case
    loose_exception_policy // assignment of out of range value should throw
>;

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

int main(int argc, const char * argv[]){
    std::cout << "example 84:\n";
    input_safe_t x, y;
```

```

try{
    std::cout << "type in values in format x y:" << std::flush;
    std::cin >> x >> y; // read variables, maybe throw exception
}
catch(const std::exception & e){
    // none of the above should trap. Mark failure if they do
    std::cout << e.what() << std::endl;
    return 1;
}
std::cout << "x" << safe_format(x) << std::endl;
std::cout << "y" << safe_format(y) << std::endl;
std::cout << safe_format(f(x, y)) << std::endl;
return 0;
}

```

- As before, we define a type `safe_t` to reflect our view of legal values for this program. This uses the `automatic` type promotion policy as well as the `loose_trap_policy` exception policy to enforce elimination of runtime penalties.
- The function `f` accepts only arguments of type `safe_t` so there is no need to check the input values. This performs the functionality of *programming by contract* with no runtime cost.
- In addition, we define `input_safe_t` to be used when reading variables from the program console. Clearly, these can only be checked at runtime so they use the `throw_exception` policy. When variables are read from the console they are checked for legal values. We need no ad hoc code to do this, as these types are guaranteed to contain legal values and will throw an exception when this guarantee is violated. In other words, we automatically get checking of input variables with no additional programming.
- On calling of the function `f`, arguments of type `input_safe_t` are converted to values of type `safe_t`. In this particular example, it can be determined at compile time that construction of an instance of a `safe_t` from an `input_safe_t` can never fail. Hence, no `try/catch` block is necessary. The usage of the `loose_trap_policy` policy for `safe_t` types guarantees this to be true at compile time.

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

```

example 84:
type in values in format x y:33 45
x<signed char>[-24,82] = 33
y<signed char>[-24,82] = 45
z = <short>[-48,164] = 78
(x + y) = <short>[-48,164] = 78
(x - y) = <signed char>[-106,106] = -12
<short>[-48,164] = 78

```

3.2. Composition with Other Libraries

For many years, Boost has included a library to represent and operate on `rational numbers`. Its well crafted, has good documentation and is well maintained. Using the rational library is as easy as construction an instance with the expression `rational r(n, d)` where `n` and `d` are integers of the same type. From then on it can be used pretty much as any other number. Reading the documentation with safe integers in mind one finds

Limited-precision integer types [such as `int`] may raise issues with the range sizes of their allowable negative values and positive values. If the negative range is larger, then the extremely-negative numbers will not have an additive inverse in the positive range, making them unusable as denominator values since they cannot be normalized to positive values (unless the user is lucky enough that the input components are not relatively prime pre-normalization).

Which hints of trouble. Examination of the code reveals which suggest that care has been taken implement operations so as to avoid overflows, catch divide by zero, etc. But the code itself doesn't seem to consistently implement this idea. So we make a small demo program:

```
#include <iostream>
#include <limits>

#include <boost/rational.hpp>
#include <safe_integer.hpp>

int main(int argc, const char * argv[]){
    // simple demo of rational library
    const boost::rational<int> r {1, 2};
    std::cout << "r = " << r << std::endl;
    const boost::rational<int> q {-2, 4};
    std::cout << "q = " << q << std::endl;
    // display the product
    std::cout << "r * q = " << r * q << std::endl;

    // problem: rational doesn't handle integer overflow well
    const boost::rational<int> c {1, INT_MAX};
    std::cout << "c = " << c << std::endl;
    const boost::rational<int> d {1, 2};
    std::cout << "d = " << d << std::endl;
    // display the product - wrong answer
    std::cout << "c * d = " << c * d << std::endl;

    // solution: use safe integer in rational definition
    using safe_rational = boost::rational<
        boost::numeric::safe<int>
    >;

    // use rationals created with safe_t
    const safe_rational sc {1, INT_MAX};
    std::cout << "c = " << sc << std::endl;
    const safe_rational sd {1, 2};
    std::cout << "d = " << sd << std::endl;
    std::cout << "c * d = ";
    try {
        // multiply them. This will overflow
        std::cout << sc * sd << std::endl;
    }
    catch (std::exception const& e) {
        // catch exception due to multiplication overflow
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

which produces the following output

```
r = 1/2
q = -1/2
r * q = -1/4
c = 1/2147483647
d = 1/2
c * d = 1/-2
```

```
c = 1/2147483647
d = 1/2
c * d = multiplication overflow: positive overflow error
```

The [rational library documentation](#) is quite specific as to the [type requirements](#) placed on the underlying type. Turns out the our [own definition of an integer type](#) fulfills (actually surpasses) these requirements. So we have reason to hope that we can use `safe<int>` as the underlying type to create what we might call a "safe_rational". This we have done in the above example where we demonstrate how to compose the rational library with the safe numerics library in order to create what amounts to a safe_rational library - all without writing a line of code! Still, things are perfect though. Since the [rational numbers library](#) implements its own checking for divide by zero by throwing an exception, the safe numerics code for handling this - included exception policy will not be respected. To summarize:

- In some cases safe types can be used as template parameters to other types to inject the concept of "no erroneous results" into the target type.
- Such composition is not guaranteed to work. The target type must be reviewed in some detail.

4. Background

This library started out as a re-implementation of the facilities provided by [David LeBlanc's SafeInt Library](#). I found this library to be well done in every way. My main usage was to run unit tests for my embedded systems projects on my PC. Still, I had a few issues.

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

This version addresses these issues. It exploits many facilities of C++14 and the [Boost libraries](#) to reduce the number of lines of source code to approximately 4700.

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 a specialization of `std::numeric_limits<T>`. See the documentation for the standard library class `numeric_limits`. The standard library includes such specializations for all the built-in 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 Numeric
---------	-----------------------------------

<code>t, u</code>	An object of a type modeling Numeric
<code>os</code>	An object of type <code>std::base_ostream</code>
<code>is</code>	An object of type <code>std::base_istream</code>

Associated Types

<code>std::numeric_limits<T></code>	The <code>numeric_limits</code> 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](#) 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
<code>std::numeric_limits<T>::is_bounded</code>	true or false
<code>std::numeric_limits<T>::is_specialized</code>	true or false
<code>std::numeric_limits<T>::is_integer</code>	true or false
<code>std::numeric_limits<T>::is_signed</code>	true or false
<code>os << t</code>	<code>os &</code>
<code>is >> t</code>	<code>is &</code>

Table 1.2. Unary Operators

Expression	Return Type	Semantics
<code>-t</code>	T	Invert sign
<code>+t</code>	T	unary plus - a no op
<code>t--</code>	T	post decrement
<code>t++</code>	T	post increment
<code>--t</code>	T	pre decrement
<code>++t</code>	T	pre increment

Table 1.3. Binary Operators

Expression	Return Type	Semantics
<code>t - u</code>	V	subtract u from t
<code>t + u</code>	V	add u to t
<code>t * u</code>	V	multiply t by u
<code>t / u</code>	T	divide t by u
<code>t % u</code>	T	t modulus u
<code>t < u</code>	bool	true if t less than u, false otherwise
<code>t <= u</code>	bool	true if t less than or equal to u, false otherwise
<code>t > u</code>	bool	true if t greater than u, false otherwise
<code>t >= u</code>	bool	true if t greater than or equal to u, false otherwise
<code>t == u</code>	bool	true if t equal to u, false otherwise
<code>t != u</code>	bool	true if t not equal to u, false otherwise
<code>t = u</code>	T	assign value of u to t
<code>t += u</code>	T	add u to t and assign to t
<code>t -= u</code>	T	subtract u from t and assign to t
<code>t *= u</code>	T	multiply t by u and assign to t
<code>t /= u</code>	T	divide t by u and assign to t

Models

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

Header

```
#include <boost/numeric/safe_numerics/concepts/numeric.hpp>
```

5.2. Integer<T>

Description

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

More specifically, a type T is Integer if there exists a 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 built-in numeric types. Note that this concept is distinct from the C++ standard library type traits `is_integral` and `is_arithmetic`. These latter fulfill the requirements 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](#)

Notation

<code>T, U, V</code>	A type that is a model of the Integer
<code>t, u</code>	An object of type modeling Integer

Valid Expressions

In addition to the expressions defined in [Numeric](#) the following expressions must be valid.

Table 1.4. General

Expression	Value
<code>std::numeric_limits<T>::is_integral</code>	<code>true</code>

Expression	Return Type	Semantics
<code>~t</code>	<code>T</code>	bitwise complement
<code>t << u</code>	<code>T</code>	shift <code>t</code> left <code>u</code> bits
<code>t >> u</code>	<code>T</code>	shift <code>t</code> right by <code>u</code> bits
<code>t & u</code>	<code>V</code>	and of <code>t</code> and <code>u</code> padded out to max # bits in <code>t, u</code>
<code>t u</code>	<code>V</code>	or of <code>t</code> and <code>u</code> padded out to max # bits in <code>t, u</code>
<code>t ^ u</code>	<code>V</code>	exclusive or of <code>t</code> and <code>u</code> padded out to max # bits in <code>t, u</code>
<code>t <<= u</code>	<code>T</code>	left shift the value of <code>t</code> by <code>u</code> bits
<code>t >>= u</code>	<code>T</code>	right shift the value of <code>t</code> by <code>u</code> bits
<code>t &= u</code>	<code>T</code>	and the value of <code>t</code> with <code>u</code> and assign to <code>t</code>
<code>t = u</code>	<code>T</code>	or the value of <code>t</code> with <code>u</code> and assign to <code>t</code>
<code>t ^= u</code>	<code>T</code>	exclusive or the value of <code>t</code> with <code>u</code> and assign to <code>t</code>

Models

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

Header

```
#include <boost/numeric/safe_numerics/concepts/integer.hpp>
```

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 they are guaranteed not to produce invalid arithmetic results. These operations return safe types rather than built-in types.

Refinement of

[Numeric](#) or [Integer](#)

Notation

Symbol	Description
T, U	Types fulfilling Numeric or Integer type requirements.
t, u	objects of types T, U
S	A type fulfilling SafeNumeric type requirements
$s, s1, s2$	objects of types S
op	C++ infix operator supported by underlying type T
$prefix_op$	C++ prefix operator: $-$, $+$, \sim , $++$, $--$ supported by underlying type T
$postfix_op$	C++ postfix operator: $++$, $--$ supported by underlying type T
$assign_op$	C++ assignment operator

Valid Expressions

Expression	Result Type	Description
$s \ op \ t$	unspecified S	invoke C++ operator op and return another SafeNumeric type.
$t \ op \ s$	unspecified S	invoke C++ operator op and return another SafeNumeric type.
$s1 \ op \ s2$	unspecified S	invoke C++ operator op and return another SafeNumeric type.
$prefix_op \ S$	unspecified S	invoke C++ operator $prefix_op$ and return another SafeNumeric type.

Expression	Result Type	Description
<code>S postfix_op</code>	unspecified S	invoke C++ operator <code>postfix_op</code> and return another SafeNumeric type.
<code>s assign_op t</code>	<code>S &</code>	convert <code>t</code> to type <code>S</code> and assign it to <code>s</code> .
<code>t assign_op s</code>	<code>T &</code>	convert <code>s</code> to type <code>T</code> and assign it to <code>s</code> . If the value <code>t</code> cannot be represented as an instance of type <code>S</code> , it is an error.
<code>S(t)</code>	<code>S</code>	construct an instance of <code>S</code> from a value of type <code>T</code> . In this case, <code>T</code> is referred to as the base type of <code>S</code> . If the value <code>t</code> cannot be represented as an instance of type <code>S</code> , it is an exception condition is invoked.
<code>S</code>	<code>S</code>	construct an uninitialized instance of <code>S</code> .
<code>T(s)</code>	<code>T</code>	implicit conversion of the value of <code>s</code> to type <code>T</code> . If the value of <code>s</code> cannot be correctly represented as a type <code>T</code> , an exception condition is invoked.
<code>static_cast<T>(s)</code>	<code>T</code>	convert the value of <code>s</code> to type <code>T</code> . If the value of <code>s</code> cannot be correctly represented as a type <code>T</code> , an exception condition is invoked.
<code>is_safe<S></code>	<code>std::true_type</code>	type trait to query whether any type <code>S</code> fulfills the requirements for a SafeNumeric type.
<code>base_type<S>::type</code>	<code>T</code>	Retrieve the base type of a given safe type.
<code>base_value(s)</code>	<code>T</code>	Retrieve the value of an instance of a safe type. This is equivalent to <code>static_cast<base_type<S>>(s)</code> .

- The 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 and whose operands are both safe types require that promotion and exception policies of the operands be identical.
- Operations on safe types are supported if and only if the same operation is supported on the underlying types. For example, the binary operations `|`, `&`, `^` and `~` operations defined for safe unsigned integer types. But they are not defined for floating point types. Currently they are also defined for signed integer types. It's not clear that this is the correct decision. On one hand, usage of these operators on signed types is almost certainly an error in program logic. But trapping this as an error conflicts with the goal of making safe types "drop-in" replacements for the corresponding built-in types. In light of this, these operators are currently supported as they are for normal built-in types.
- Safe Numeric types will be implicitly converted to built-in types when appropriate. Here's an example:

```
void f(int);

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

This behavior supports the concept of `safe<T>` as being a "drop-in" replacement for a `T`.

Complexity Guarantees

There are no complexity guarantees explicitly enforced here. However, it would be very surprising if any implementation were to be more complex than $O(1)$;

Invariants

The fundamental requirement of a `SafeNumeric` type is that it implements all C++ operations permitted on its base type in a way that 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...). But no implementation should return an arithmetically incorrect result.

Models

```
safe<T>

safe_signed_range<-11, 11>

safe_unsigned_range<0, 11>

safe_signed_literal<4>
```

Header

```
#include <boost/numeric/safe_numerics/concepts/safe_numeric.hpp>
```

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 the library permits one to specify his own type promotion rules via a `PromotionPolicy` class.

Notation

PP	A type that fully fills the requirements of a <code>PromotionPolicy</code>
T, U	A type that is a model of the <code>Numeric</code> concept
R	An object of type modeling <code>Numeric</code> which can be used to construct a <code>SafeNumeric</code> type.

Valid Expressions

Any operations which result in integers which cannot be represented as some `Numeric` type will throw an exception. These expressions return a type which can be used as the basis to create a `SafeNumeric` type.

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::comparison_result<T, U>::type	bool
PP::left_shift_result<T, U>::type	unspecified Numeric type
PP::right_shift_result<T, u>::type	unspecified Numeric type
PP::bitwise_or_result<T, U>::type	unspecified Numeric type
PP::bitwise_and_result<T, U>::type	unspecified Numeric type
PP::bitwise_xor_result<T, U>::type	unspecified Numeric type

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, native> sx;
auto sz = sx + y; // standard C++ code which detects errors
```

Type `sz` will be a [SafeNumeric](#) type based on `int`. If the result exceeds the maximum value that can be stored in an `int`, an error is detected.

The `native` 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, automatic> sx;
auto sz = sx + y;
    // sz is a safe type based on long
    // hence sz is guaranteed not to overflow.
safe_unsigned_range<1, 4> a;
safe_unsigned_range<2, 4> b;
auto c = a + b; // c will be a safe type with a range [3,8] and cannot overflow

```

Type `sz` will be a [SafeNumeric](#) type which is guaranteed to hold the 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](#)

Header

```
#include <boost/numeric/safe_numerics/concepts/promotion_policy.hpp>
```

5.5. ExceptionPolicy<EP>

Description

The exception policy specifies what is to occur when a safe operation cannot return a valid 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 fulfills the requirements of an <code>ExceptionPolicy</code>
e	A code from safe_numerics_error
message	A const char * which refers to a text message about the cause of an exception

Valid Expressions

Whenever an operation yields an invalid result, one of the following functions will be invoked.

Expression	Return Value	Invoked when:
<code>EP::on_arithmetic_error(e, message)</code>	void	The operation cannot produce valid arithmetic result such as overflows, divide by zero, etc.
<code>EP::on_undefined_behavior(e, message)</code>	void	The result is undefined by the C++ standard
<code>EP::on_implementation_defined_behavior(e, message)</code>	void	The result depends upon implementation defined behavior according to the C++ standard
<code>EP::on_uninitialized_value(e, message)</code>	void	A variable is not initialized

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

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

Synopsis

```
template<class EP>
constexpr void
dispatch<EP>(const boost::numeric::safe_numerics_error & e, char const * const & msg);
```

Example of use

```
#include <boost/safe_numerics/exception_policies.hpp>

dispatch<boost::numeric::loose_exception_policy>(
    boost::numeric::safe_numerics_error::positive_overflow_error,
    "operation resulted in overflow"
);
```

Models

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

- `boost::numeric::loose_exception_policy`

Throw on arithmetic errors, ignore other errors. Some applications ignore these issues and still work and we don't want to update them.

- `boost::numeric::loose_trap_policy`

Same as above in that it doesn't check for various undefined behaviors but traps at compile time for hard arithmetic errors. This policy would be suitable for older embedded systems which depend on bit manipulation operations to work.

- `boost::numeric::strict_exception_policy`

Permit just about anything, throw at runtime on any kind of error. Recommended for new code. Check everything at compile time if possible and runtime if necessary. Trap or Throw as appropriate. Should guarantee code to be portable across architectures.

- `boost::numeric::strict_trap_policy`

Same as above but requires code to be written in such a way as to make it impossible for errors to occur. This naturally will require extra coding effort but might be justified for embedded and/or safety critical systems.

- `boost::numeric::default_exception_policy`

Alias for `strict_exception_policy`, One would use this first. After experimentation, one might switch to one of the above policies or perhaps use a custom policy.

Header

```
#include <boost/numeric/safe_numerics/concepts/exception_policy.hpp>
```

6. Types

6.1. `safe<T, PP, EP>`

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 to trap in some way.

Notation

Symbol	Description
<code>T</code>	Underlying type from which a safe type is being derived

Associated Types

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

Template Parameters

Parameter	Type Requirements	Description
<code>T</code>	Integer<T>	The underlying type. Currently only integer types are supported
<code>PP</code>	PromotionPolicy<PP>	Default value is <code>boost::numeric::native</code>
<code>EP</code>	Exception Policy<EP>	Default value is <code>boost::numeric::default_exception_policy</code>

See examples below.

Model of

[Integer](#)

[SafeNumeric](#)

Valid Expressions

Implements all expressions and only those expressions defined by the [SafeNumeric<T>](#) type requirements. Note that all these expressions are `constexpr`. Thus, 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.

When a binary operand is applied to two instances of `safe<T, PP, EP>` on the following must be true:

- The promotion policies of the two operands must be the same or one of them must be void
- The exception policies of the two operands must be the same or one of them must be void

If either of the above is not true, a compile error will result.

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 as "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 an 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. In the following example, the normal implicit conversions function the same for safe integers as they do for built-in integers.

```
#include <limits>
#include <safe_integer.hpp>
using namespace boost::numeric;

int f(int i){
    return i;
}

using safe_t = safe<long>;

int main(){
    const long x = 97;
    f(x); // OK - implicit conversion to int
    const safe_t y = 97;
    f(y); // Also OK - checked implicit conversion to int
    return 0;
}
```

When the `safe<long>` is implicitly converted to an `int` when calling `f`, the value is checked to be sure that it is within the legal range of an `int` and will invoke an exception if it cannot. We can easily verify this by altering the exception handling policy in the above example to `loose_trap_policy`. This will invoke a compile time error on any conversion might invoke a runtime exception.

```
#include <limits>
#include <safe_integer.hpp>
using namespace boost::numeric;

int f(int i){
    return i;
}

using safe_t = safe<long, native, loose_trap_policy>;

int main(){
    const long x = 97;
    f(x);    // OK - implicit conversion to int
    const safe_t y = 97;
    f(y);    // Would be OK, but will invoke compile time error
    return 0;
}
```

But this raises it's own questions. We can see that in this example, the program can never fail:

- The value 97 is assigned to `y`
- `y` is converted to an `int`
- and used as an argument to `f`

The conversion can never fail because the value of 97 can always fit into an `int`. But the library code can't detect this and emits the checking code even though it's not necessary.

This can be addressed by using a [safe_literal](#). A safe literal can contain one and only one value. All the functions in this library are marked `constexpr`. So it can be determined at compile time that conversion to an `int` can never fail and no runtime checking code need be emitted. Making this small change will permit the above example to run with zero runtime overhead while guaranteeing that no error can ever occur.

```
#include <limits>
#include <safe_integer.hpp>
#include <safe_literal.hpp>
using namespace boost::numeric;

int f(int i){
    return i;
}

template<intmax_t N>
using safe_literal = safe_signed_literal<N, native, loose_trap_policy>;

int main(){
    const long x = 97;
    f(x);    // OK - implicit conversion to int
    const safe_literal<97> y;
    f(y);    // OK - y is a type with min/max = 97;
    return 0;
}
```

```
}
```

With this trivial example, such efforts would hardly be deemed necessary. But in a more complex case, perhaps including compile time arithmetic expressions, it could be much more difficult to verify that the constant is valid and/or no checking code is needed. And there is also possibility that over the life time of the application, the compile time constants might change, thus rendering any ad hoc analyse obsolete. Using `safe_literal` will future-proof your code against well-meaning, but code-breaking updates.

Adjust type promotion rules.

Another way to avoid arithmetic errors like overflow is to promote types to larger sizes before doing the arithmetic.

Stepping back, we can see that many of the cases of invalid arithmetic wouldn't exist if the result 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 - no checking for exceptions is 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::default_exception_policy
    >;
    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
}
```

Header

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

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 to trap in some way.

Notation

Symbol	Description
<code>MIN, MAX</code>	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 a non-negative 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 <code>boost::numeric::native</code>
EP	Exception Policy<EP>	Default value is <code>boost::numeric::default_exception_policy</code>

Model of

[Integer](#)

[SafeNumeric](#)

Valid Expressions

Implements all expressions and only those expressions defined by the [SafeNumeric](#) type requirements. Thus, 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.

Example of use

```
#include <type_traits>
#include <../include/safe_integer.hpp>
#include <../include/safe_range.hpp>

#include <../include/utility.hpp>

using namespace boost::numeric;

void f(){
    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;

// if either or both types are safe types, the result is a safe type
// determined by promotion policy. In this instance
// the range of i is [7, 24] and the range of j is [0,255].
// so the type of k will be a safe type with a range of [7,279]
static_assert(
    is_safe<decltype(k)>::value
    && std::numeric_limits<decltype(k)>::min() == 7
    && std::numeric_limits<decltype(k)>::max() == 279,
    "k is a safe range of [7,279]"
);
}

```

Header

```
#include <boost/numeric/safe_numerics/safe_range.hpp>
```

6.3. `safe_signed_literal<Value, PP, EP>` and `safe_unsigned_literal<Value, PP, EP>`

Description

A safe type which holds a literal value. This is required to be able to initialize other safe types in such a way that an 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.

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
Value	Integer	value used to initialize the literal
PP	PromotionPolicy<PP>	Default value is <code>void</code>
EP	Exception Policy<EP>	Default value is <code>void</code>

Model of

[Integer](#)

[SafeNumeric](#)

[LiteralType](#)

Inherited Valid Expressions

safe literal types are immutable. Hence they only inherit those valid expressions which don't change the value. This excludes assignment, increment, and decrement and all unary operators except unary -, + and ~. Other than that, they can be used anywhere a SafeNumeric type can be used. Note that the default promotion and exception policies are void. This is usually convenient since when a safe literal is used in a binary operation, this will inherit the policies of the other type. On the other hand, this can be inconvenient when operands of a binary expression are both safe literals. This will fail to compile since there are no designated promotion and exception policies. They way to address this to assign specific policies as in this example.

```
template<typename T>
using compile_time_value = safe_signed_literal<T>;

constexpr compile_time_value<1000> x;
constexpr compile_time_value<0> y;

// should compile and execute without problem

std::cout << x << '\n';

// all the following statements should fail to compile
constexpr safe<int> z = x / y;
```

New Valid Expression

Expression	Result Type	Description
<code>t op s</code>	unspecified S	invoke C++ operator <code>op</code> and return another SafeNumeric type.
<code>s1 op s2</code>	unspecified S	invoke C++ operator <code>op</code> and return another SafeNumeric type.
<code>prefix_op S</code>	unspecified S	invoke C++ operator <code>prefix_op</code> and return another SafeNumeric type.
<code>S postfix_op</code>	unspecified S	invoke C++ operator <code>postfix_op</code> and return another SafeNumeric type.
<code>s assign_op t</code>	S1	convert <code>t</code> to type <code>S</code> and assign it to <code>s</code> . If the value <code>t</code> cannot be represented as an instance of type <code>S</code> , it is an error.
<code>S(t)</code>	unspecified S	construct an instance of <code>S</code> from a value of type <code>T</code> . If the value <code>t</code> cannot be represented as an instance of type <code>S</code> , it is an error.
<code>S</code>	S	construct an uninitialized instance of <code>S</code> .
<code>is_safe<S></code>	<code>std::true_type</code> or <code>std::false_type</code>	type trait to query whether any type <code>S</code> fulfills the requirements for a SafeNumeric type.
<code>static_cast<T>(s)</code>	T	convert the value of <code>s</code> to type <code>T</code> . If the value of <code>s</code> cannot be correctly represented as a type <code>T</code> , it is an error.

Example of use

```
#include <boost/numeric/safe_numerics/safe_literal.hpp>
```



```
constexpr boost::numeric::safe_signed_literal<42> x;
```

Header

```
#include <boost/numeric/safe_numerics/safe_literal.hpp>
```

6.4. exception

Description

Here we describe the data types used to refer to exceptional conditions which might occur. Note that when we use the word "exception", we don't mean the C++ term which refers to a data type, but rather the colloquial sense of a anomaly, irregularity, deviation, special case, isolated example, peculiarity, abnormality, oddity; misfit, aberration or out of the ordinary occurrence. This concept of "exception" is more complex than one would think and hence is not manifested by a single simple type. A small number of types work together to implement this concept within the library.

We've leveraged on the `std::error_code` which is part of the standard library. We don't use all the facilities that it offers but it's not an exact match, but it's useful for our purposes.

enum class safe_numerics_error

The following values are those which a numeric result might return. They resemble the standard error codes used by C++ standard exceptions. This resemblance is coincidental and they are wholly unrelated to any codes of similar names. The reason for the resemblance is that the library started its development using the standard library codes. But as development progressed it became clear that the original codes weren't sufficient so now they stand on their own. Here are a list of error codes. The description of what they mean is

Symbol	Description
success	successful operation - no error returned
positive_overflow_error	A positive number is too large to be represented by the data type
negative_overflow_error	The absolute value of a negative number is too large to be represented by the data type.
underflow_error	A number is too close to zero to be represented by the data type.
range_error	an argument to a function or operator is outside the legal range - e.g. <code>sqrt(-1)</code> .
domain_error	the result of an operation is outside the legal range of the result.
implementation_defined_behavior	operation may or may not return the expected result.
undefined_behavior	According to the C++ standard, the result is undefined. e.g. <code>10 << 80</code> shifts bits off the left and returns an unexpected result of 0 on most machines.
uninitialized_value	According to the C++ standard, the result may be defined by the application. e.g. <code>16 >> 10</code> will result the expected result of 0 on most machines.

The above listed codes can be transformed to an instance of type `std::error_code` with the function:

```
std::error_code make_error_code(safe_numerics_error e)
```

This object can be

enum class safe_numerics_actions

The above error codes are classified into groups according to how such exceptions should be handled. The following table shows the possible actions that an error could be mapped to.

Symbol	Description
no_action	successful operation - no action action required
uninitialized_value	report attempt to use an uninitialized value - not currently used
arithmetic_error	report an arithmetic error
implementation_defined_behavior	report an operation which the C++ standard permits but fails to specify
undefined_behavior	report an operation whose result is undefined by the C++ standard.

Translation of a `safe_numerics_error` into the corresponding `safe_numerics_action` can be accomplished with the following function:

```
constexpr enum safe_numerics_actions  
make_safe_numerics_action(const safe_numerics_error & e);
```

See Also

- [C++ Standard Library version](#) The C++ standard error handling utilities.
- [Thinking Asynchronously in C++](#) Another essential reference on the design and usage of the `error_code`

Header

```
#include <boost/numeric/safe_numerics/exception.hpp>
```

6.5. exception_policy<AE, IDB, UB, UV>

Description

Create a valid exception policy from 4 function objects. This specifies the actions to be taken for different types of invalid results.

Template Parameters

Parameter	Type Requirements	Invoked when:
AE	Function object callable with the expression <code>AE(e, message)</code>	The operation cannot produce valid arithmetic result such as overflows, divide by zero, etc.
UB	Function object callable with the expression <code>UB(e, message)</code>	The result is undefined by the C++ standard

Parameter	Type Requirements	Invoked when:
IDB	Function object callable with the expression IDB(e,	The result depends upon implementation defined behavior according to the C++ standard
UV	Function object callable with the expression UV(e, message)	A variable is not initialized

Model of

[ExceptionPolicy](#)

Inherited Valid Expressions

This class implements all the valid operations from the type requirements [ExceptionPolicy](#). Aside from these, there are no other operations implemented.

Function Objects

In order to create an exception policy, one needs some function objects. The library includes some appropriate examples of these:

Name	Description
ignore_exception	Ignore any runtime exception and just return - thus propagating the error. This is what would happen with unsafe data types
throw_exception	throw an exception of type <code>std::system_error</code>
trap_exception	Invoke a function which is undefined. Compilers will include this function if and only if there is a possibility of a runtime error. Conversely, This will create a compile time error if there is any possibility that the operation will fail at runtime. Use the action to guarantee that your application will never produce an invalid result. Any operation invoke

But of course one is free to provide his own. Here is an example of a function object which would could be used exception conditions.

```
// log an exception condition but continue processing as though nothing has happened
// this would emulate the behavior of an unsafe type.
struct log_runtime_exception {
    log_runtime_exception(const safe_numerics_error & e, const char * message){
        std::cout << "Caught system_error with code " << e.code()
                   << " meaning " << e.what() << '\n';
    }
};
```

Policies Provided by the library

The above function object can be composed into an exception policy by this class. The library provides common policies all ready to use. In the table below, the word "loose" is used to indicate that implementation defined and undefined behavior is not considered an exceptional condition, while "strict" means the opposite. The word "exception" means that a runtime exception will be thrown. The word "trap" means that the mere possibility of an error condition will result in a compile time error.

Name	Description
<code>loose_exception_policy</code>	Throws runtime exception on any arithmetic error. Undefined and implementation defined behavior is permitted as long as it does not produce an arithmetic error.
<code>loose_trap_policy</code>	Invoke a compile time error in any case where it's possible to result in an arithmetic error.
<code>strict_exception_policy</code>	Throws runtime exception on any arithmetic error. Any undefined or implementation defined behavior also results in throwing an exception.
<code>strict_trap_policy</code>	Invoke a compile time error in any case where it's possible to result in an arithmetic error, undefined behavior or implementation defined behavior
<code>default_exception_policy</code>	an alias for <code>strict_exception_policy</code>

If none of the above suit your needs, you're free to create your own. Here is one where use the logging function object defined above as a component in a loose exception policy which logs any arithmetic errors and ignores any other types of errors.

```
// logging policy
// log arithmetic errors but ignore them and continue to execute
// implementation defined and undefined behavior is just executed
// without logging.

using logging_exception_policy = exception_policy<
    log_runtime_exception,    // arithmetic error
    ignore_exception,         // implementation defined behavior
    ignore_exception,         // undefined behavior
    ignore_exception          // uninitialized value
>;
```

Header

```
#include <boost/numeric/safe_numerics/concept/exception_policy.hpp>
```

6.6. Promotion Policies

native

Description

This type contains the functions to return a safe type corresponding to the C++ type resulting from 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. Its 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;
}
```

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 its default promotion parameter so explicitly using `native` is not necessary.

```
#include <cassert>
#include <boost/numeric/safe_numerics/safe_integer.hpp>
#include <boost/numeric/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 an invalid result an exception is thrown
        z = x + y;
        assert(false); // never arrive here
    }
    catch(std::exception & e){
        // which we can catch here
        std::cout << e.what() << std::endl;
    }

    // When result is an int, C++ promotion rules guarantee
    // 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 guaranteed correct without any runtime overhead or exception.

    return 0;
}
```

Notes

See Chapter 5, Expressions, C++ Standard

Header

```
#include <boost/numeric/safe_numerics/native.hpp>
```

automatic

Description

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

The standard C/C++ procedure for executing arithmetic operations on operands of different types is:

- Convert operands to some common type using a somewhat elaborate elaborate rules defined in the C++ standard.
- Execute the operation.
- If the result of the operation cannot fit in the common type of the operands, discard the high order bits.

The automatic promotion policy replaces the standard C/C++ procedure for the following one:

- Convert operands to some common type using to the following rules.
 - For addition. If the operands are both unsigned the common type will be unsigned. Otherwise it will be signed.
 - For subtraction, the common type will be signed.
 - For left/right shift, the sign of the result will be the sign of the left operand.
 - For all other types of operands, if both operands are unsigned the common type will be unsigned. Otherwise, it will be signed.
- Determine the smallest size of the signed or unsigned type which can be guaranteed hold the result.
- If this size exceeds the maximum size supported by the compiler, use the maximum size supported by the compiler.
- Execute the operation.
 - Convert each operand to the common type.
 - If the result cannot be contained in the result type as above, invoke an error procedure.
 - Otherwise, return the result in the common type

This type promotion policy is applicable only to safe types whose base type is an [Integer](#) type.

Model of

[PromotionPolicy](#)

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;
    // z is guaranteed correct without any runtime overhead or exception.
    auto z = x + y;
    return 0;
}
```

Header

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

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

Description

This policy is used to promote safe types in arithmetic expressions according to the 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 programs 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 desktop.

Note that this policy is only applicable to safe types whose base type is a type fulfilling the type requirements of [Integer](#).

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](#)

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 in the 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

// emulate environment 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::loose_trap_policy // 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 the "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.

Header

```
#include <boost/numeric/safe_numerics/cpp.hpp>
```

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. This information has been included to help those who want to understand how the library works so they can extend it, correct bugs in it, or understand its limitations. These components are also interesting in their own right. For all these reasons, they are documented here. In general terms, the library works in the following manner:

- All unary/binary expressions where one of the operands is a "safe" type are overloaded. These overloads are declared and defined in the header file "safe_integer.hpp". SFINAE - "Substitution Failure Is Not An Error" and `std::enable_if` are key features of C++ used to define these overloads in a correct manner.
- Each overloaded operation implements the following:
 - Retrieve range of values for each operand of type T from `std::numeric_limits<T>::min()` and `std::numeric_limits<T>::max()`.
 - Given the ranges of the operands, determine the range of the result of the operation using interval arithmetic. This is implemented in the "interval.hpp" header file using the `constexpr` facility of C++14.
 - If the range of the result type includes the range of the result of the operation, no run time checking of the result is necessary. So the operation reduces to the original built-in C/C++ operation.
 - Otherwise, the operation is implemented as a "checked integer operation" at run time. This operation returns a variant which will contain either a correct result or an exception enum indicating why a correct result could not be obtained. The variant object is implemented in the header file "checked_result.hpp" and the checked operations are implemented in "checked.hpp".
 - If a valid result has been obtained, it is passed to the caller.
 - Otherwise, an exception is invoked.

The following components realize the design described here. Although they have been crafted specifically to support this library, there is no reason that that they cannot be used on their own. In fact, several of them very much lend themselves to such usage.

8.1. `checked_result<R>`

Description

`checked_result` is a wrapper class designed to hold the 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 is not explicitly exposed to users. But it is important to know about this if one wants to extend the application of the library to other types. It may be useful in other applications.

Template Parameters

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

Parameter	Description
<code>R</code>	Type of value expected if there is not error to be reported

Notation

Symbol	Description
<code>R</code>	Type of value expected if there is not error to be reported
<code>r</code>	An instance of type <code>R</code>
<code>c</code>	an instance of <code>checked_result<R></code>
<code>t</code>	an instance of <code>checked_result<T></code> for some type <code>T</code> not necessarily the same as <code>R</code>
<code>e</code>	An instance of type <code>safe_numerics_error</code>

Valid Expressions

All expressions are `constexpr`.

Expression	Return Type	Semantics
<code>checked_result(r)</code>	<code>checked_result<R></code>	constructor with valid instance of <code>R</code>
<code>checked_result<R>(t)</code>	<code>checked_result<R></code>	constructor with <code>checked_result<T></code> where <code>T</code> is not <code>R</code> . <code>T</code> must be convertible to <code>R</code> .
<code>checked_result(e, msg)</code>	<code>checked_result<R></code>	constructor with error information
<code>static_cast<R>(c)</code>	<code>R</code>	extract wrapped value - asserts if not possible

Expression	Return Type	Semantics
<code>static_cast<e>(c)</code>	<code>safe_numerics_error</code>	extract wrapped value - asserts if not possible
<code>static_cast<const char *>(c)</code>	<code>const char *</code>	returns pointer to error message
<code>c < t</code> <code>c >= t</code> <code>c > t</code> <code>c <= t</code> <code>c == t</code> <code>c != t</code>	<code>boost::logic::tribool</code>	compare the wrapped values of two <code>checked_result</code> instances. If either one contains an <code>error_type</code> value, return <code>boost::logic::tribool::indeterminant</code> .
<code>c.exception()</code>	<code>bool</code>	true if <code>checked_result</code> contains an error condition

Example of use

```
#include <boost/numeric/safe_numerics/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](#)

Header

```
#include <boost/numeric/safe_numerics/checked_result.hpp>
```

8.2. Checked Arithmetic

Synopsis

```
// safe casting on primitive types
template<class R, class T>
checked_result<R> constexpr checked::cast(const T & t);

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

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

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

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

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

// safe less than predicate on primitive types
template<class R, class T, class U>
bool constexpr checked::less_than(const T & t, const U & u);

// safe greater_than_equal predicate on primitive types
template<class R, class T, class U>
bool constexpr checked::greater_than_equal(const T & t, const U & u);

// safe greater_than predicate on primitive types
template<class R, class T, class U>
bool constexpr checked::greater_than(const T & t, const U & u);

// safe less_than_equal predicate on primitive types
template<class R, class T, class U>
bool constexpr checked::less_than_equal(const T & t, const U & u);

// safe equal predicate on primitive types
template<class R, class T, class U>
bool constexpr checked::equal(const T & t, const U & u);

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

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

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

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

template<class R, class T, class U>
checked_result<R> constexpr 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.

Example of use

```
#include <boost/numeric/safe_numerics/checked.hpp>

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

Notes

Some compilers have command line switches (e.g. `-ftrapv`) which enable special behavior such that 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<R>](#)

Header

```
#include <boost/numeric/safe_numerics/checked.hpp>
```

```
#include <boost/numeric/safe_numerics/checked.hpp>
```

```
#include <boost/numeric/safe_numerics/checked.hpp>
```

8.3. 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 of [Numeric](#)

Notation

Symbol	Description
I	An interval type
i, j	An instance of interval type

Symbol	Description
R	Numeric types which can be used to make an interval
l, u	Lowermost and uppermost values in an interval
os	std::basic_ostream<Char, CharT>

Associated Types

<code>checked_result</code>	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
<code>interval<R>(l, u)</code>	<code>interval<R></code>	construct a new interval from a pair of limits
<code>interval<R>(p)</code>	<code>interval<R></code>	construct a new interval from a pair of limits
<code>interval<R>(i)</code>	<code>interval<R></code>	copy constructor
<code>i.l</code>	R	lowermost value in the interval i
<code>i.u</code>	R	uppermost value in the interval i
<code>i.includes(j)</code>	bool	return true if interval i includes interval j
<code>i.includes(t)</code>	bool	return true if interval i includes value t
<code>add<R>(i, j)</code>	<code>interval<checked_result<R>></code>	add two intervals and return the result
<code>subtract<R>(i, j)</code>	<code>interval<checked_result<R>></code>	subtract two intervals and return the result
<code>multiply<R>(i, j)</code>	<code>interval<checked_result<R>></code>	multiply two intervals and return the result
<code>divide<R>(i, j)</code>	<code>interval<checked_result<R>></code>	divide one interval by another and return the result
<code>modulus<R>(i, j)</code>	<code>interval<checked_result<R>></code>	calculate modulus of one interval by another and return the result
<code>left_shift<R>(i, j)</code>	<code>interval<checked_result<R>></code>	calculate the range that would result from shifting one interval by another
<code>right_shift<R>(i, j)</code>	<code>interval<checked_result<R>></code>	calculate the range that would result from shifting one interval by another
<code>bitwise_or<R>(i, j)</code>	<code>interval<checked_result<R>></code>	range of values which can result from applying to any pair of operands from I and j

Expression	Return Type	Semantics
<code>bitwise_and<R>(i, j)</code>	<code>interval<checked_result<R>></code>	range of values which can result from applying <code>&</code> to any pair of operands from I and j
<code>bitwise_xor<R>(i, j)</code>	<code>interval<checked_result<R>></code>	range of values which can result from applying <code>^</code> to any pair of operands from I and j
<code>t < u</code>	<code>boost::logic::tribool</code>	true if every element in t is less than every element in u
<code>t > u</code>	<code>boost::logic::tribool</code>	true if every element in t is greater than every element in u
<code>t <= u</code>	<code>boost::logic::tribool</code>	true if every element in t is less than or equal to every element in u
<code>t >= u</code>	<code>boost::logic::tribool</code>	true if every element in t is greater than or equal to every element in u
<code>t == u</code>	<code>bool</code>	true if limits are equal
<code>t != u</code>	<code>bool</code>	true if limits are not equal
<code>os << i</code>	<code>os &</code>	print interval to output stream

Example of use

```
#include <iostream>
#include <cstdint>
#include <cassert>
#include <boost/numeric/safe_numerics/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;
}
```

Header

```
#include <boost/numeric/safe_numerics/interval.hpp>
```

8.4. `safe_compare<T, U>`

Synopsis

```
// compare any pair of integers
template<class T, class U>
bool constexpr safe_compare::less_than(const T & lhs, const U & rhs);
```

```
template<class T, class U>
bool constexpr safe_compare::greater_than(const T & lhs, const U & rhs);

template<class T, class U>
bool constexpr safe_compare::less_than_equal(const T & lhs, const U & rhs);

template<class T, class U>
bool constexpr safe_compare::greater_than_equal(const T & lhs, const U & rhs);

template<class T, class U>
bool constexpr safe_compare::equal(const T & lhs, const U & rhs);

template<class T, class U>
bool constexpr safe_compare::not_equal(const T & lhs, const U & rhs);
```

Description

Compare two primitive integers. These functions will return a correct result regardless of the type of the operands. Specifically it is guaranteed to return the correct arithmetic result when comparing signed and unsigned types of any size. It does not follow the standard C/C++ procedure of converting the operands to some common type then doing the compare. So it is not equivalent to the C/C++ binary operations `<`, `>`, `>=`, `<=`, `==`, `!=`. The functions are free functions defined inside the namespace `boost::numeric::safe_compare`.

Type requirements

All template parameters of the functions must be C/C++ built-in integer types, `char`, `int`

Complexity

Each function performs one and only one arithmetic operation.

Example of use

```
#include <cassert>
#include <safe_compare.hpp>

using namespace boost::numeric;
const short int x = -64;
const unsigned int y = 42000;

assert(x < y); // fails
assert(safe_compare::less_than(x, y)); // OK
```

Header

```
#include <boost/numeric/safe_numerics/safe_compare.hpp>
```

9. Performance Tests

Our goal is to create facilities which make it possible to 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 taken precautions to avoid doing this, the only real way to know is to create and run some tests.

So far we've only run one explicit performance test - [test_performance.cpp](#). This runs a test from the Boost Multiprecision library to count prime numbers and uses on integer arithmetic. We've run the tests with unsigned integers and with `safe<unsigned>` on two different compilers.. No other change was made to the program. We list the results without further comment.

```
g++ (GCC) 6.2.0
Testing type unsigned:
time = 17.6215
count = 1857858
Testing type safe<unsigned>:
time = 22.4226
count = 1857858

clang-802.0.41
Testing type unsigned:
time = 16.9174
count = 1857858
Testing type safe<unsigned>:
time = 36.5166
count = 1857858
```

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 think that it's a widely held view. The best answer is to consider the examples in the [Tutorials and Motivating Examples](#) section of the library documentation.

- 10.2.** Can safe types be used as drop-in replacements 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. Occasionally compile time errors will occur and adjustments to the source code will be required. Typically these will result in code which is more correct.

- 10.3.** Why are there special types for literal such as `safe_signed_literal<42>`? Why not just use `std::integral_const<int, 42>`?

By defining our own "special" type we can simplify the interface. Using `std::integral_const` requires one to specify both the type *and* the value. Using `safe_signed_literal<42>` doesn't require a parameter for the type. So the library can select the best type to hold the specified value. It also means that one won't have the opportunity to specify a type-value pair which are inconsistent.

- 10.4.** Why is `safe...literal` needed at all? What's the matter with `const safe<int>(42)`?

`const safe<int>(42)` looks like it might be what we want: An immutable value which invokes the "safe" operators when used in an expression. But there is one problem. The `std::numeric_limits<safe<int>>` is a range from `INTMIN` to `INTMAX` even though the value is fixed to 42 at compile time. It is this range which is used at compile time to calculate the range of the result of the operation.

So when an operation is performed, the range of the result is calculated from `[INTMIN, INTMAX]` rather than from `[42,42]`.

- 10.5.** Are safe type operations `constexpr`? That is, can they be invoked at compile time?

Yes. safe type construction and calculations are all `constexpr`. Note that to get maximum benefit, you'll have to use `safe...literal` to specify the independent values at compile time.

10.6. Why define `safe_literal`? Isn't it effectively the same as `std::integral_constant`?

Almost, but there are still good reasons to create a different type.

- `std::integral_constant<int, 42>` requires specification of type as well as value so it's less convenient than `safe_signed_literal` which maps to the smallest type required to hold the value.
- `std::numeric_limits<std::integral_constant<int, 42>>::is_integer` returns `false`. This would complicate implementation of the library
- type trait `is_safe<std::integral_constant<int, 42>>` would have to be defined to return `true`.
- But globally altering the traits of `std::integral_constant` might have unintended side-effects related to other code. These might well be surprises which are create errors which are hard to find and hard to work around.

10.7. Why is `Boost.Convert` not used?<

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

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

I used "safe" in large part because 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.

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

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

10.10. Isn't putting a defensive check just before any potential undefined behavior 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]

10.11. 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 complement. However, this has yet to be verified in a rigorous way.

10.12. Why is it necessary to specialize `numeric_limits` for "safe" types?

`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 work with `safe<T>`. `safe<T>` is a drop-in replacement for `T` so it has to implement all the operations.

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

The guiding purpose of the library is to trap incorrect arithmetic behavior - not just undefined behavior. Although a savvy user may understand and keep present in his mind that an unsigned integer is really a modular type, the plain reading of an arithmetic expression conveys the idea that all operands are integers. Also in many cases, `unsigned integers` are used

in cases where modular arithmetic is not intended, such as array indices. 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 be a representation of a subset of integers. Note that this decision is consistent with [INT30-C], “Ensure that unsigned integer operations do not wrap” in the CERT C Secure Coding Standard [Seacord].

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

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

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

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

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

```
#ifdef TEST
// using C++ on test platform
#include <cstdint>
#include <boost/numeric/safe_numerics/safe_integer.hpp>
#include <cpp.hpp>
using pic16_promotion = boost::numeric::cpp<
    8, // char
    8, // short
    8, // int
    16, // long
    32 // long long
>;
// define safe types used in the desktop version of the program.
template <typename T> // T is char, int, etc data type
using safe_t = boost::numeric::safe<
    T,
    pic16_promotion,
    boost::numeric::default_exception_policy // use for compiling and running tests
>;
typedef safe_t<std::int_least16_t> int16_t;
typedef safe_t<std::int_least32_t> int32_t;
#else
/* using C on embedded platform */
typedef int int_least16_t;
typedef long int_least16_t;
#endif
```

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

- Compile for the target C platform with the macro TEST undefined.

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

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 has been taken 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 all other possible C++ 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. Acknowledgements

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

David LeBlanc

This library is inspired by [David LeBlanc's SafeInt Library](#) . I found this library very well done in every way and useful in my embedded systems work. This motivated me to take it to the "next level".

[Andrzej Krzemienski](#)

Andrzej Commented and reviewed the library as it was originally posted on the [Boost Library Incubator](#). The consequent back and forth motivated me to invest more effort in developing documentation and examples to justify the utility, indeed the necessity, for this library. He also noted many errors in code, documentation, and tests. Without his interest and effort, I do not believe the library would have progressed beyond its initial stages.

[Boost](#)

As always, the Boost Developer's mailing list has been the source of many useful observations from potential users and constructive criticism from very knowledgeable developers. During the Boost formal review, reviews and comments were posted by the following persons:

- Paul A. Bristow
- Steven Watanabe
- John Maddock
- Antony Polukhin
- Barrett Adair
- Vicente J. Botet Escriba

- John McFarlane
- Peter Dimov

13. Change Log

This is the third version.

14. Bibliography

Bibliography

- [coker] Zack Coker, Samir Hasan, Jeffrey Overbey, Munawar Hafiz, Christian Kästner. *Integers In C: An Open Invitation To Security Attacks?*. JTC1/SC22/WG21 - The C++ Standards Committee - ISO CPP. January 15, 2012. Coker
- [crowl1] Lawrence Crowl. *C++ Binary Fixed-Point Arithmetic*. JTC1/SC22/WG21 - The C++ Standards Committee - ISO CPP. January 15, 2012. Crowl
- [crowl2] Lawrence Crowl, Thorsten Ottosen. *Proposal to add Contract Programming to C++*. WG21/N1962 and J16/06-0032 - The C++ Standards Committee - ISO CPP. February 25, 2006. Crowl & Ottosen
- [dietz] Will Dietz, Peng Li, John Regehr, Vikram Adve. *Understanding Integer Overflow in C/C++*. Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland. June 2012. Dietz
- [garcia] J. Daniel Garcia. *C++ language support for contract programming*. WG21/N4293 - The C++ Standards Committee - ISO CPP. December 23, 2014. Garcia
- [katz] Omer Katz. *SafeInt code proposal*. Boost Developer's List. Katz
- [keaton] David Keaton, Thomas Plum, Robert C. Seacord, David Svoboda, Alex Volkovitsky, and Timothy Wilson. *As-if Infinitely Ranged Integer Model*. Software Engineering Institute. CMU/SEI-2009-TN-023.
- [leblanc1] David LeBlanc. *Integer Handling with the C++ SafeInt Class*. Microsoft Developer Network. January 7, 2004. LeBlanc
- [leblanc2] David LeBlanc. *SafeInt*. CodePlex. Dec 3, 2014. LeBlanc
- [lions] Jacques-Louis Lions. *Ariane 501 Inquiry Board report*. Wikisource. July 19, 1996. Lions
- [matthews] Hubert Matthews. *CheckedInt: A Policy-Based Range-Checked Integer*. Overload Journal #58. December 2003. Matthews
- [mouawad] Jad Mouawad. *F.A.A Orders Fix for Possible Power Loss in Boeing 787*. New York Times. April 30, 2015.
- [plakosh] Daniel Plakosh. *Safe Integer Operations*. U.S. Department of Homeland Security. May 10, 2013. Plakosh
- [seacord1] Robert C. Seacord. *Secure Coding in C and C++*. 2nd Edition. Addison-Wesley Professional. April 12, 2013. 978-0321822130. Seacord
- [seacord2] Robert C. Seacord. *INT30-C. Ensure that operations on unsigned integers do not wrap*. Software Engineering Institute, Carnegie Mellon University. August 17, 2014. INT30-C
- [seacord3] Robert C. Seacord. *INT32-C. Ensure that operations on signed integers do not result in overflow*. Software Engineering Institute, Carnegie Mellon University. August 17, 2014. INT32-C

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

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