A Proposal to Add Safe Integer Types to the Standard Library Technical Report

Document number:	P0228R0
Project:	Programming Language C++
Audience:	Library Evolution Working Group
Author:	Robert Ramey
Contact:	ramey@rrsd.com
Date:	2016-02-16

1 Motivation

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

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

2 Impact On the Standard

This proposal is a pure library extension. It does not require changes to any standard classes, functions or headers. It might benefit from relaxing some of the conditions on aggregate types. It has been implemented in and requires standard C++/14.

3 Design Decisions

The template class is designed to function as closely as possible as a drop-in replacement for corresponding built-in integer types.

1. "Drop In Replacement for Any Integer Type"

   The template class is designed to function as closely as possible as a drop-in replacement for corresponding built-in integer types. Ideally, one should be able to just substitute safe<T> for all instances of T in any program and expect it compile and execute as before with no other changes.

Since C++ permits freely mixing signed and unsigned integer types in expressions, safe versions of these types can also be. This complicates the implementation of the library to significant degree.

2. "Return No Incorrect Results"

   Usage of a safe type in a binary expression is guaranteed to either return an arithmetically correct result or throw a standard exception.

3. "Automatically Inter operate with built-in integer types"

   The usage of a safe type in binary expression "infects" the expression by returning another safe type. This is designed to avoid accidentally losing the safety of the expression.

4. "Uses <limits> instead of type traits"

   Implementation of a library such as this necessarily keeps track of the types of data objects. The most common way to do this is using type_traits such as `std::is_integral`, `std::is_unsigned`, `std::is_arithmetic`, etc. This doesn't work very well for a few reasons:

   These are defined by the standard to apply only to built-in types. Specializing these traits for new types such as safe<int> would conflict with the standard.

   We are allowed to create specialization of std::numeric_limits for our own types - including safe<T>. So this works well for us.

   safe<T> might be implemented in such as way that it would work for unforeseen integer-like types such as "money". Numeric limits has more complete information about these types which might make it easier to extend the library.

5. "Performance"

   Performance will depend on the implementation and subject to the constraints above. This design will permit the usage of template meta-programming to eliminate runtime performance penalties in some cases. In the following example, there is no runtime penalty required to guarantee that incorrect results will never be generated.

```
#include <cstdint>
```

```
#include <safe>

using namespace std;

int f(safe<int8_t> i){
    // C++ promotion rules make overflow on multiplication impossib
    // cannot fail on return
    // zero performance penalty
    return i * i;
}

int8_t f(safe<int8_t> i){
    // C++ promotion rules make overflow on multiplication impossib
    // but result could be truncated on return
    // so result must be checked at runtime incurring a runtime pen
    return i * i;      // cannot overflow on multiplication,
}
```

Some processors have the ability to detect erroneous results but the C++ language doesn't include the ability to exploit these features. Implementor's of this library will have the option to exploit these features to diminish or eliminate runtime costs.

If all else fails and the runtime cost is deemed too large for the program to bear, users will have the option of creating their own aliases for the types the program uses and assign them according to the whether they are building a "Debug" or "Release" version. This is not ideal, but would still be preferable to the current approach which generally consists of ignoring the possibility that C++ numeric operations may produce arithmetically incorrect results.

6. "No Extra Parameters"

   An alternative to this proposal would be a policy based design which would permit users to select or define actions to be taken in the case of errors. This is quite possible and likely useful. However, the simplicity usage of the current proposal is an important feature. So I decided not to include it.

7. "No other safe types"

   Other ideas come to mind such as `safe<Min, Max>`, `safe_literal<Value>`, and others. I excluded these in the spirit of following the controlling purpose of making a "drop in replacement". Once one included these types into a program, they change the semantics of the program so that it's not really C++ any more. There is a place for these ideas, (see below), but I don't think the standard library is that place.

4 Existing Implementations

This proposal is a simpler version / subset of the Safe Numerics library in development by Robert Ramey on the Boost Library Incubator. It is compatible with this proposal but it also includes:

Policy classes for error handling

Policy classes for type promotion. These permit substitution of C++ standard type promotion rules with other ones which can reduce or eliminate the need for runtime error checking code.

Other safe types such as safe_integer_range<Min, Max>.

Complete documentation including internal operation

Without comment, here are implementations of libraries which are in some way similar to this proposal

Robert Leahy, Safe integer utilities for C++11

David LeBlanc, SafeInt

David Stone, Bounded Integer

5 Technical Specifications

5.1 Type Requirements

5.1.1 Numeric<T>

5.1.1.1 Description

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

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

5.1.1.2 Notation

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

5.1.1.3 Associated Types

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

5.1.1.4 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.

General

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

Unary Operators

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

Binary Operators

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

`t |= u`	T	or the value of t with u and assign to t
`t ^= u`	T	exclusive or the value of t with u and assign to t

5.1.1.5 Header

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

5.1.1.6 Models

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

The definition of this concept

5.1.2 Integer<T>

5.1.2.1 Description

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

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

5.1.2.2 Refinement of

Numeric

5.1.2.3 Valid Expressions

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

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

5.1.2.4 Header

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

5.1.2.5 Models

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

5.1.3 SafeNumeric<T>

5.1.3.1 Description

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

5.1.3.2 Refinement of

Numeric

5.1.3.3 Notation

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

5.1.3.4 Valid Expressions

Expression	Result Type	Description
`s op t`	unspecified S	invoke safe C++ operator op and return anoth SafeNumeric type.
`t op s`	unspecified S	invoke safe C++ operator op and return anoth SafeNumeric type.
`s1 op s2`	unspecified S	invoke safe C++ operator op and return anoth SafeNumeric type.
`prefix_op S`	unspecified S	invoke safe C++ operator op and return anoth SafeNumeric type.
`S postfix_op`	unspecified S	invoke safe C++ operator op and return anoth SafeNumeric type.

`s assign_op t`	S1	convert t to type S1 and assign it to s1. If the v cannot be represented as an instance of type S an error.
`S(t)`	unspecified S	construct a instance of S from a value of type T value t cannot be represented as an instance o S1, it is an error.
`S`	S	construct a uninitialized instance of S.
`is_safe<S>`	`std::true_type` or `std::false_type`	type trait to query whether any type T fulfills tl requirements for a SafeNumeric type.
`static_cast<T>(s)`	T	convert the value of s to type T. If the value of cannot be correctly represented as a type T, it i error.

- Result of any binary operation where one or both of the operands is a SafeNumeric type is also a SafeNumeric type.
- All the expressions in the above table are `constexpr` expressions
- Binary expressions which are not assignments require that promotion and exception policies be identical.

5.1.3.5 Complexity Guarantees

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

5.1.3.6 Invariants

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

5.1.3.7 Header

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

5.1.3.8 Models

safe<T>

safe_signed_range<-11, 11>

safe_unsigned_range<0, 11>

safe_literal<4>

6 Types

6.1 safe<T>

6.1.1 Description

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

6.1.2 Notation

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

6.1.3 Template Parameters

Parameter	Type Requirements	Description
`T`	Integer	The underlying type. Currently only integer types supported

See examples below.

6.1.4 Model of

Integer

SafeNumeric

6.1.5 Valid Expressions

Implements all expressions defined by the SafeNumeric type requirements.

`safe<T>` is meant to be a "drop-in" replacement of the intrinsic integer types.

The type of an expression of type safe<T> op safe<U> will be safe<R> where R would be the same as the type of the expression T op U.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 std::exception will be thrown.

6.1.6 Header

`#include <safe>`

6.1.7 Example of use

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

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

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

7 Acknowledgements

This proposal is a simplified version of Safe Numeics library proposed for Boost. This effort was inspired by David LeBlanc's SafeInt Library .

8 References

Author	Omer Katz
Title	SafeInt code proposal
Publishername	Boost Developer's List
Abbrev	Katz
Abstract	Posts of various authors regarding a proposed SafeInt library for boost

Author	David LeBlanc

Title	Integer Handling with the C++ SafeInt Class
Publishername	Microsoft Developer Network
Date	January 7, 2004
Abbrev	LeBlanc

Author	David LeBlanc
Title	SafeInt
Publishername	CodePlex
Date	Dec 3, 2014
Abbrev	LeBlanc

Author	Jacques-Louis Lions
Title	Ariane 501 Inquiry Board report
Publishername	Wikisource
Date	July 19, 1996
Abbrev	Lions

Author	Daniel Plakosh
Title	Safe Integer Operations
Publishername	U.S. Department of Homeland Security
Date	May 10, 2013
Abbrev	Plakosh

Author	Robert C. Seacord
Title	Secure Coding in C and C++
Edition	2nd Edition
Publishername	Addison-Wesley Professional
Date	April 12, 2013
Isbn	978-0321822130
Abbrev	Seacord

Author	Robert C. Seacord

Title	INT30-C. Ensure that operations on unsigned integers do not wrap
Publishername	Software Engineering Institute, Carnegie Mellon University
Date	August 17, 2014
Abbrev	INT30-C

Author	Robert C. Seacord
Title	INT32-C. Ensure that operations on signed integers do not result in overflow
Publishername	Software Engineering Institute, Carnegie Mellon University
Date	August 17, 2014
Abbrev	INT32-C