Chapter 1. Safe Numerics

Robert Ramey

Copyright © 2012-2018 Robert Ramey

Table of Contents

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 unexpectedly 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 are 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 to be arithmetically correct or invoke an error. Using this library, the above example would be rendered as:

```
#include <boost/safe_numerics/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. Arithmetic expressions will not 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 "safe" types are

meant to be "drop-in" replacements for the built-in types of the same name. 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 only 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 compile 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 violate these requirements will be trapped at either compile time 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

• mp11

• integer

• config

• tribool

- enable_if

The Safe Numerics library is delivered with an exhaustive suite of test programs.

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++ programs 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 built in integer types with corresponding safe types.

```
//  Copyright (c) 2018 Robert Ramey
//
// Distributed under the Boost Software License, Version 1.0. (See
// accompanying file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

#include <iostream>

#include <boost/safe_numerics/safe_integer.hpp>

int main(int, const char *[]){
    std::cout << "example 1:";
    std::cout << "undetected erroneous expression evaluation" << std::endl;
    std::cout << "Not using safe numerics" << std::endl;
    try{
        std::int8_t x = 127;
        std::int8_t y = 2;
        std::int8_t z;
        // this produces an invalid result !
        z = x + y;
        std::cout << "error NOT detected!" << std::endl;
        std::cout << (int)z << " != " << (int)x << " + " << (int)y << std::endl;
    }
    catch(const 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::safe_numerics;
        safe<std::int8_t> x = INT_MAX;
```

```
        safe<std::int8_t> y = 2;
        safe<std::int8_t> z;
        // rather than producing an invalid result an exception is thrown
        z = x + y;
    }
    catch(const 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
error NOT detected!
-127 != 127 + 2
Using safe numerics
error detected:converted signed value too large: positive overflow error
Program ended with exit code: 0
```

2.2. Arithmetic Operations Can Overflow Silently

A variation of the above is when a value is incremented/decremented beyond its domain.

```
//  Copyright (c) 2018 Robert Ramey
//
// Distributed under the Boost Software License, Version 1.0. (See
// accompanying file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

#include <iostream>

#include <boost/safe_numerics/safe_integer.hpp>

int main(int, const char *[]){
    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(const 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::safe_numerics;
        safe<int> x = INT_MAX;
        // throws exception when result is past maximum possible
        ++x;
        assert(false); // never arrive here
```

```
    }
    catch(const 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. Arithmetic on Unsigned Integers Can Yield Incorrect Results

Subtracting two unsigned values of the same size will result in an unsigned value. If the first operand is less than the second the result will be arithmetically in correct. But if the size of the unsigned types is less than that of an `unsigned int`, C/C++ will promote the types to `signed int` before subtracting resulting in an correct result. In either case, there is no indication of an error. Somehow, the programmer is expected to avoid this behavior. Advice usually takes the form of "Don't use unsigned integers for arithmetic". This is well and good, but often not practical. C/C++ itself uses unsigned for `sizeof(T)` which is then used by users in arithmetic.

This program demonstrates this problem. The solution is to replace instances of built in integer types with corresponding safe types.

```
//  Copyright (c) 2018 Robert Ramey
//
// Distributed under the Boost Software License, Version 1.0. (See
// accompanying file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

#include <iostream>

#include <boost/safe_numerics/safe_integer.hpp>

int main(int, const char *[]){
    std::cout << "example 8:";
    std::cout << "undetected erroneous expression evaluation" << std::endl;
    std::cout << "Not using safe numerics" << std::endl;
    try{
        unsigned int x = 127;
        unsigned int y = 2;
        unsigned int z;
        // this produces an invalid result !
        z = y - x;
        std::cout << "error NOT detected!" << std::endl;
        std::cout << z << " != " << y << " - " << x << std::endl;
    }
```

```
    catch(const 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::safe_numerics;
        safe<unsigned int> x = 127;
        safe<unsigned int> y = 2;
        safe<unsigned int> z;
        // rather than producing an invalid result an exception is thrown
        z = y - x;
        std::cout << "error NOT detected!" << std::endl;
        std::cout << z << " != " << y << " - " << x << std::endl;
    }
    catch(const std::exception & e){
        // which we can catch here
        std::cout << "error detected:" << e.what() << std::endl;
    }
    return 0;
}
```

```
example 8:undetected erroneous expression evaluation
Not using safe numerics
error NOT detected!
4294967171 != 2 - 127
Using safe numerics
error detected:subtraction result cannot be negative: negative overflow error
Program ended with exit code: 0
```

2.4. 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:

```
//  Copyright (c) 2018Robert Ramey
//
// Distributed under the Boost Software License, Version 1.0. (See
// accompanying file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

#include <iostream>

#include <boost/safe_numerics/safe_integer.hpp>

int main(){
    std::cout << "example 4: ";
    std::cout << "implicit conversions change data values" << std::endl;
    std::cout << "Not using safe numerics" << std::endl;

    // problem: implicit conversions change data values
    try{
        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(const 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::safe_numerics;
        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(const 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, usage of the safe types results in no runtime overhead in using the safe integer library. Code generated will either equal or exceed the efficiency of using primitive integer types.

2.5. 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:

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

#include <boost/safe_numerics/safe_integer.hpp>

using namespace std;
using namespace boost::safe_numerics;

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: mixing types produces surprising 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>`.

2.6. 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 <array>

#include <boost/safe_numerics/safe_integer_range.hpp>

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

int main(int, const char *[]){
    // 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::safe_numerics;
```

```
        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(const 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
```

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.

2.7. 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 <boost/safe_numerics/safe_integer.hpp>

int main(int, const char *[]){
    // 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(const 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::safe_numerics;
        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(const 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.

2.8. 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 exception_policy selected by the library user.

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

#include <boost/safe_numerics/safe_integer.hpp>

int main(int, const char *[]){
    // 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 cannot be handled at runtime!" << std::endl;
    }
```

```
    catch(const std::exception &){
        std::cout << "error handled at runtime!" << std::endl;
    }
    // solution: replace int with safe<int>
    std::cout << "Using safe numerics" << std::endl;
    try{
        using namespace boost::safe_numerics;
        const safe<int> x = 1;
        const safe<int> y = 0;
        std::cout << x / y;
        std::cout << "error NOT detected!" << std::endl;
    }
    catch(const std::exception & e){
        std::cout << "error handled at runtime!" << 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
```

2.9. Compile Time Arithmetic is Not Always Correct

If a divide by zero error occurs while a program is being compiled, there is not guarantee that it will be detected. This example shows a real example compiled with a recent version of CLang.

• Source code includes a constant expression containing a simple arithmetic error.

• The compiler emits a warning but otherwise calculates the wrong result.

• Replacing int with safe<int> will guarantee that the error is detected at runtime

• Operations using safe types are marked constexpr. So we can force the operations to occur at runtime by marking the results as constexpr. This will result in an error at compile time if the operations cannot be correctly calculated.

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

#include <boost/safe_numerics/safe_integer.hpp>

int main(int, const char *[]){
    // problem: cannot recover from arithmetic errors
    std::cout << "example 8: ";
    std::cout << "cannot detect compile time arithmetic errors" << std::endl;
    std::cout << "Not using safe numerics" << std::endl;

    try{
        const int x = 1;
        const int y = 0;
        // will emit warning at compile time
        // will leave an invalid result at runtime.
        std::cout << x / y; // will display "0"!
        std::cout << "error NOT detected!" << std::endl;
    }
```

```
    catch(const 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::safe_numerics;
        const safe<int> x = 1;
        const safe<int> y = 0;
        // constexpr const safe<int> z = x / y; // note constexpr here!
        std::cout << x / y; // error would be detected at runtime
        std::cout << " error NOT detected!" << std::endl;
    }
    catch(const std::exception & e){
        std::cout << "error detected:" << e.what() << std::endl;
    }
    return 0;
}
```

```
example 8: cannot detect compile time arithmetic errors
Not using safe numerics
0error NOT detected!
Using safe numerics
error detected:positive overflow error
Program ended with exit code: 0
```

2.10. 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 <boost/safe_numerics/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 numerics to enforce program contract automatically
// define convenient typenames for hours and minutes hh:mm
using hours_t = boost::safe_numerics::safe_unsigned_range<0, 23>;
using minutes_t = boost::safe_numerics::safe_unsigned_range<0, 59>;
using minutes_total_t = boost::safe_numerics::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 guarentees corectness
        // 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, const char *[]){
        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(const 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(const 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. Eliminating Runtime Penalty

Up until now, we've mostly 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 cost of checking many arithmetic operations at runtime. 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 <boost/safe_numerics/safe_integer.hpp>
#include <boost/safe_numerics/exception_policies.hpp> // include exception policies

using safe_t = boost::safe_numerics::safe<
    int,
    boost::safe_numerics::native,
    boost::safe_numerics::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 compile. There are a couple of ways to achieve this.

3.1. Using safe_range and safe_literal

When trying to avoid arithmetic errors of the above type, programmers will select data types which are wide enough to hold values large enough to be certain that results won't overflow, but are not so large as to make the program needlessly inefficient. 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.

```cpp
#include <iostream>

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

using namespace boost::safe_numerics;

// 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, const char *[]){
    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;
    const safe_t z = x + y;
    std::cout << "x + y = " << safe_format(x + y) << std::endl;
    std::cout << "z = " << safe_format(z) << std::endl;
    return 0;
}
```

- `safe_signed_range` defines a type which is limited to the indicated range. Out of range assignments will be detected at compile time if possible (as in this case) or at run time if necessary.

- 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 at compile time.

- 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.

The output uses a custom output manipulator, `safe_format`, for safe types to display the underlying type and its range as well as current value. This program produces the following run time output.

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

- The sum x + y can also be determined at compile time.

- 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 + y because it is in the range that `z` is guaranteed to hold. If the sum could not be 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.

All this information regarding the range and 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 - although it might 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.

3.2. 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 for arithmetic operations are 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.

When using the `automatic` type promotion policy, with a 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 <boost/safe_numerics/safe_integer.hpp>
#include <boost/safe_numerics/exception_policies.hpp>
#include <boost/safe_numerics/automatic.hpp>
#include "safe_format.hpp" // prints out range and value of any type

using safe_t = boost::safe_numerics::safe<
    int,
    boost::safe_numerics::automatic, // note use of "automatic" policy!!!
    boost::safe_numerics::loose_trap_policy
>;

int main(int, const char *[]){
    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 << "x + y = " << 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. This is because since a `long` can always hold the result of the sum of two integers.

- 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
x + y = <long>[-4294967296,4294967294] = 2147483649
```

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`.

Note that once we use automatic type promotion, our programming language isn't C/C++ anymore. So don't be tempted to so something like the following:

```
// DON'T DO THIS !
#if defined(NDEBUG)
using safe_t = boost::numeric::safe<
    int,
    boost::numeric::automatic, // note use of "automatic" policy!!!
    boost::numeric::loose_trap_policy
>;
#else
using safe_t = boost::numeric::safe<int>;
#endif
```

3.3. Mixing Approaches

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

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

#include <boost/safe_numerics/safe_integer.hpp>
#include <boost/safe_numerics/safe_integer_range.hpp>
#include <boost/safe_numerics/automatic.hpp>
#include <boost/safe_numerics/exception.hpp>

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

using namespace boost::safe_numerics;
```

```
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 varibles, maybe throw exception
    }
    catch(const std::exception & e){
        // none of the above should trap. Mark failure if they do
        std::cout << e.what() << std::endl;
        return 1;
    }
    std::cout << "x" << safe_format(x) << std::endl;
    std::cout << "y" << safe_format(y) << std::endl;
    std::cout << safe_format(f(x, y)) << std::endl;
    return 0;
}
```

- As before, we define a type `safe_t` to reflect our view of legal values for this program. This uses 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
```

4. Case Studies

4.1. 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:

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

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

int main(int, const char *[]){
    // 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::safe_numerics::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 not perfect. 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.

4.2. Safety Critical Embedded Controller

Suppose we are given the task of creating stepper motor driver software to drive a robotic hand to be used in robotic micro surgery. The processor that has been selected by the engineers is the PIC18F2520 manufactured by Microchip Corporation. This processor has 32KB of program memory. On a processor this small, it's common to use a mixture of 8, 16, and 32 bit data types in order to minimize memory footprint and program run time. The type int has 16 bits. It's programmed in C. Since this program is going to be running life critical function, it must be demonstrably correct. This implies that it needs to be verifiable and testable. Since the target micro processor is inconvenient for accomplishing these goals, we will build and test the code on the desktop.

How a Stepper Motor Works

Figure 1.1: Stepper Motor receives a pulse which causes the motor to move one step. It seems simple, but in practice it turns out
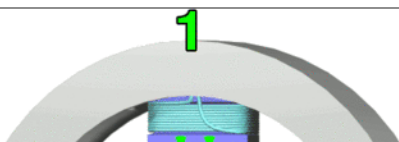
to be quite intricate to get right as one has to time the pulses individually to smoothly accelerate the rotation of the motor from a standing start until it reaches the some maximum velocity. Failure to do this will either limit the stepper motor to very low speed or result in skipped steps when the motor is under load. Similarly, a loaded motor must be slowly decelerated down to a stop.

Figure 1.2. Motion Profile
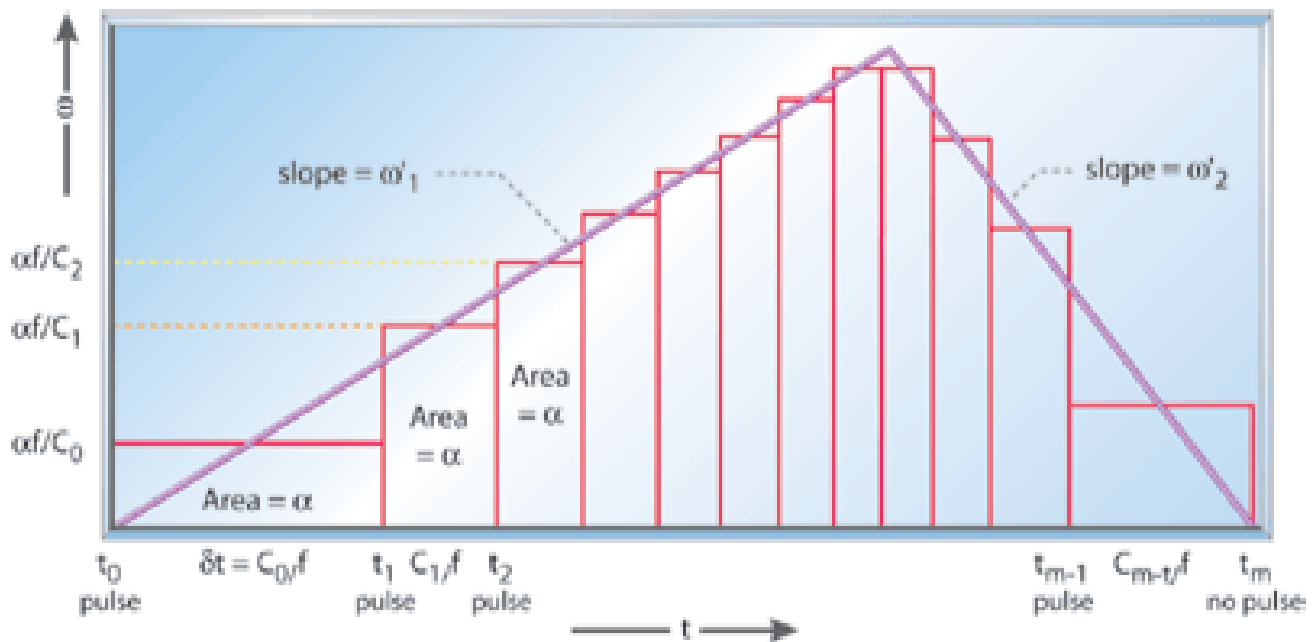
This implies the the width of the pulses must decrease as the motor accelerates. That is the pulse with has to be computed while the motor is in motion. This is illustrated in the above drawing. A program to accomplish this might look something like the following:

setup registers and step to zero position

specify target position
set initial time to interrupt
enable interrupts

On interrupt
   if at target position
     disable interrupts and return
   calculate width of next step
   change current winding according to motor direction
   set delay time to next interrupt to width of next step

Already, this is turning it to a much more complex project than it first seemed. Searching around the net, we find a popular article on the operation of stepper motors using simple micro controllers. The algorithm is very well explained and it includes complete code we can test. The engineers are still debugging the prototype boards and hope to have them ready before the product actually ships. But this doesn't have to keep us from working on our code.

Updating the Code

Inspecting this code, we find that it is written in a dialect of C rather than C itself. At the time this code was written, conforming versions of the C compiler were not available for PIC processors. We want to compile this code on the Microchip XC8 compiler which, for the most part, is standards conforming. So we made the following minimal changes:

- Factor into motor1.c which contains the motor driving code and motor_test1.c which tests that code.

- Include header <xc.h> which contains constants for the PIC18F2520 processor

- Include header <stdint.h> to include standard Fixed width integer types.

- Include header <stdbool.h> to include keywords true and false in a C program.

- The original has some anomalies in the names of types. For example, int16 is assumed to be unsigned. This is an artifact of the original C compiler being used. So type names in the code were altered to standard ones while retaining the intent of the original code.

- Add in missing make16 function.

- Format code to personal taste.

- Replaced enable_interrupts and disable_interrupts functions with appropriate PIC commands.

The resulting program can be checked to be identical to the original but compiles on with the Microchip XC8 compiler. Given a development board, we could hook it up to a stepper motor, download and boot the code and verify that the motor rotates 5 revolutions in each direction with smooth acceleration and deceleration. We don't have such a board yet, but the engineers have promised a working board real soon now.

Refactor for Testing

In order to develop our test suite and execute the same code on the desktop and the target system we factor out the shared code as a separate module which will used in both environments without change. The shared module motor2.c contains the algorithm for handling the interrupts in such a way as to create the smooth acceleration we require.

```
motor_test2.c     example92.cpp

#include ...      #include ...
PIC typedefs ...  desktop types ...
      \          /
       \        /
     #include motor2.c
       /        \
      /          \
PIC test code     desktop test code
```

Compiling on the Desktop

Using the target environment to run tests is often very difficult or impossible due to limited resources. So software unit testing for embedded systems is very problematic and often skipped. The C language on our desktop is the same used by the PIC18F2520. So now we can also run and debug the code on our desktop machine. Once our code passes all our tests, we can download the code to the embedded hardware and run the code natively. Here is a program we use on the desktop to do that:

```
//////////////////////////////////////////////////////////////
// example92.cpp
//
// Copyright (c) 2015 Robert Ramey
//
```

```
// Distributed under the Boost Software License, Version 1.0. (See
// accompanying file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

#include <iostream>

// **************************
// 1. include headers to support safe integers
#include <boost/safe_numerics/cpp.hpp>
#include <boost/safe_numerics/exception.hpp>
#include <boost/safe_numerics/safe_integer.hpp>

// **************************
// 2. specify a promotion policy to support proper emulation of
// PIC types on the desktop
using pic16_promotion = boost::safe_numerics::cpp<
    8,  // char      8 bits
    16, // short    16 bits
    16, // int      16 bits
    16, // long     16 bits
    32  // long long 32 bits
>;

// 1st step=50ms; max speed=120rpm (based on 1MHz timer, 1.8deg steps)
#define C0    (50000 << 8)
#define C_MIN  (2500 << 8)

static_assert(C0 < 0xffffff, "Largest step too long");
static_assert(C_MIN > 0, "Smallest step must be greater than zero");
static_assert(C_MIN < C0, "Smallest step must be smaller than largest step");

// **************************
// 3. define PIC integer type names to be safe integer types of he same size.

template <typename T> // T is char, int, etc data type
using safe_t = boost::safe_numerics::safe<
    T,
    pic16_promotion
>;

// alias original program's integer types to corresponding PIC safe types
// In conjunction with the promotion policy above, this will permit us to
// guarantee that the resulting program will be free of arithmetic errors
// introduced by C expression syntax and type promotion with no runtime penalty

typedef safe_t<int8_t> int8;
typedef safe_t<int16_t> int16;
typedef safe_t<int32_t> int32;
typedef safe_t<uint8_t> uint8;
typedef safe_t<uint16_t> uint16;
typedef safe_t<uint32_t> uint32;

// **************************
// 4. emulate PIC features on the desktop

// filter out special keyword used only by XC8 compiler
#define __interrupt
// filter out XC8 enable/disable global interrupts
#define ei()
#define di()
```

```cpp
// emulate PIC special registers
uint8 RCON;
uint8 INTCON;
uint8 CCP1IE;
uint8 CCP2IE;
uint8 PORTC;
uint8 TRISC;
uint8 T3CON;
uint8 T1CON;

uint8 CCPR2H;
uint8 CCPR2L;
uint8 CCPR1H;
uint8 CCPR1L;
uint8 CCP1CON;
uint8 CCP2CON;
uint8 TMR1H;
uint8 TMR1L;

// create type used to map PIC bit names to
// correct bit in PIC register
template<typename T, std::int8_t N>
struct bit {
    T & m_word;
    constexpr explicit bit(T & rhs) :
        m_word(rhs)
    {}
    constexpr bit & operator=(int b){
        if(b != 0)
            m_word |= (1 << N);
        else
            m_word &= ~(1 << N);
        return *this;
    }
    constexpr operator int () const {
        return m_word >> N & 1;
    }
};

// define bits for T1CON register
struct  {
    bit<uint8, 7> RD16{T1CON};
    bit<uint8, 5> T1CKPS1{T1CON};
    bit<uint8, 4> T1CKPS0{T1CON};
    bit<uint8, 3> T1OSCEN{T1CON};
    bit<uint8, 2> T1SYNC{T1CON};
    bit<uint8, 1> TMR1CS{T1CON};
    bit<uint8, 0> TMR1ON{T1CON};
} T1CONbits;

// define bits for T1CON register
struct  {
    bit<uint8, 7> GEI{INTCON};
    bit<uint8, 5> PEIE{INTCON};
    bit<uint8, 4> TMR0IE{INTCON};
    bit<uint8, 3> RBIE{INTCON};
    bit<uint8, 2> TMR0IF{INTCON};
    bit<uint8, 1> INT0IF{INTCON};
    bit<uint8, 0> RBIF{INTCON};
```

```
} INTCONbits;

// ***************************
// 5. include the environment independent code we want to test
#include "motor2.c"

#include <chrono>
#include <thread>

// round 24.8 format to microseconds
int32 to_microseconds(uint32 t){
    return (t + 128) / 256;
}

using result_t = uint8_t;
const result_t success = 1;
const result_t fail = 0;

// move motor to the indicated target position in steps
result_t test(int32 m){
    try {
        std::cout << "move motor to " << m << '\n';
        motor_run(m);
        std::cout
        << "step #" << ' '
        << "delay(us)(24.8)" << ' '
        << "delay(us)" << ' '
        << "CCPR" << ' '
        << "motor position" << '\n';
        do{
            std::this_thread::sleep_for(std::chrono::microseconds(to_microseconds(c)));
            uint32 last_c = c;
            uint32 last_ccpr = ccpr;
            isr_motor_step();
            std::cout
            << step_no << ' '
            << last_c << ' '
            << to_microseconds(last_c) << ' '
            << std::hex << last_ccpr << std::dec << ' '
            << motor_pos << '\n';
        }while(run_flg);
    }
    catch(const std::exception & e){
        std::cout << e.what() << '\n';
        return fail;
    }
    return success;
}

int main(){
    std::cout << "start test\n";
    result_t result = success;
    try{
        initialize();
        // move motor to position 1000
        result &= test(1000);
        // move motor to position 200
        result &= test(200);
        // move motor to position 200 again! Should result in no movement.
        result &= test(200);
```

```
        // move back to position 0
        result &= test(0);
        // ***************************
        // 6. error detected here!  data types can't handle enough
        // steps to move the carriage from end to end! Suppress this
        // test for now.
        // move motor to position 50000.
 //      result &= test(50000);
        // move motor back to position 0.
        result &= test(0);
    }
    catch(const std::exception & e){
        std::cout << e.what() << '\n';
        return 1;
    }
    catch(...){
        std::cout << "test interrupted\n";
        return EXIT_FAILURE;
    }
    std::cout << "end test\n";
    return result == success ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Here are the essential features of the desktop version of the test program.

1. Include headers required to support safe integers.

2. Specify a promotion policy to support proper emulation of PIC types on the desktop.

   The C language standard doesn't specify sizes for primitive data types like int. They can and do differ between environments. Hence, the characterization of C/C++ as "portable" languages is not strictly true. Here we choose aliases for data types so that they can be defined to be the same in both environments. But this is not enough to emulate the PIC18F2520 on the desktop. The problem is that compilers implicitly convert arguments of C expressions to some common type before performing arithmetic operations. Often, this common type is the native int and the size of this native type is different in the desktop and embedded environment. Thus, many arithmetic results would be different in the two environments.

   But now we can specify our own implicit promotion rules for test programs on the development platform that are identical to those on the target environment! So unit testing executed in the development environment can now provide results relevant to the target environment.

3. Define PIC integer type aliases to be safe integer types of he same size.

   Code tested in the development environment will use safe numerics to detect errors. We need these aliases to permit the code in motor2.c to be tested in the desktop environment. The same code run in the target system without change.

4. Emulate PIC features on the desktop.

   The PIC processor, in common with most micro controllers these days, includes a myriad of special purpose peripherals to handle things like interrupts, USB, timers, SPI bus, I^2C bus, etc.. These peripherals are configured using special 8 bit words in reserved memory locations. Configuration consists of setting particular bits in these words. To facilitate configuration operations, the XC8 compiler includes a special syntax for setting and accessing bits in these locations. One of our goals is to permit the testing of the identical code with our desktop C++ compiler as will run on the micro controller. To realize this goal, we create some C++ code which implements the XC8 C syntax for setting bits in particular memory locations.

5. include motor1.c

6. Add test to verify that the motor will be able to keep track of a position from 0 to 50000 steps. This will be needed to maintain the position of out linear stage across a range from 0 to 500 mm.

Our first attempt to run this program fails by throwing an exception from motor1.c indicating that the code attempts to left shift a negative number at the statements:

```
denom = ((step_no - move) << 2) + 1;
```

According to the C/C++ standards this is implementation defined behavior. But in practice with all modern platforms (as far as I know), this will be equivalent to a multiplication by 4. Clearly the intent of the original author is to "micro optimize" the operation by substituting a cheap left shift for a potentially expensive integer multiplication. But on all modern compilers, this substitution will be performed automatically by the compiler's optimizer. So we have two alternatives here:

- Just ignore the issue.

  This will work when the code is run on the PIC. But, in order to permit testing on the desktop, we need to inhibit the error detection in that environment. With safe numerics, error handling is determined by specifying an exception policy. In this example, we've used the default exception policy which traps implementation defined behavior. To ignore this kind of behavior we could define our own custom exception policy.

- change the `<< 2` to `* 4`. This will produce the intended result in an unambiguous, portable way. For all known compilers, this change should not affect runtime performance in any way. It will result in unambiguously portable code.

- Alter the code so that the expression in question is never negative. Depending on sizes of the operands and the size of the native integer, this expression might return convert the operands to int or result in an invalid result.

Of these alternatives, the third seems the more definitive fix so we'll choose that one. We also decide to make a couple of minor changes to simplify the code and make mapping of the algorithm in the article to the code more transparent. With these changes, our test program runs to the end with no errors or exceptions. In addition, I made a minor change which simplifies the handling of floating point values in format of 24.8. This results in motor2.c which makes the above changes. It should be easy to see that these two versions are otherwise identical.

Finally our range test fails. In order to handle the full range we need, we'll have to change some data types used for holding step count and position. We won't do that here as it would make our example too complex. We'll deal with this on the next version.

Trapping Errors at Compile Time

We can test the same code we're going to load into our target system on the desktop. We could build and execute a complete unit test suite. We could capture the output and graph it. We have the ability to make are code much more likely to be bug free. But:

- This system detects errors and exceptions on the test machine - but it fails to address and detect such problems on the target system. Since the target system is compiles only C code, we can't use the exception/error facilities of this library at runtime.

- Testing shows the presence, not the absence of bugs. Can we not prove that all integer arithmetic is correct?

- For at least some operations on safe integers there is runtime cost in checking for errors. In this example, this is not really a problem as the safe integer code is not included when the code is run on the target - it's only a C compiler after all. But more generally, using safe integers might incur an undesired runtime cost.

Can we catch all potential problems at compiler time and therefore eliminate all runtime cost?

Our first attempt consists of simply changing default exception policy from the default runtime checking to the compile time trapping one. Then we redefine the aliases for the types used by the PIC to use this exception policy.

```
// generate compile time errors if operation could fail
using trap_policy = boost::numeric::loose_trap_policy;
...
typedef safe_t<int8_t, trap_policy> int8;
...
```

When we compile now, any expressions which could possibly fail will be flagged as syntax errors. This occurs 11 times when compiling the motor2.c program. This is fewer than one might expect. To understand why, consider the following example:

```
safe<std::int8_t> x, y;
...
safe<std::int16_t> z = x + y;
```

C promotion rules and arithmetic are such that the z will always contain an arithmetically correct result regardless of what values are assigned to x and y. Hence there is no need for any kind of checking of the arithmetic or result. The Safe Numerics library uses compile time range arithmetic, C++ template multiprogramming and other techniques to restrict invocation of checking code to only those operations which could possible fail. So the above code incurs no runtime overhead.

Now we have 11 cases to consider. Our goal is to modify the program so that this number of cases is reduced - hopefully to zero. Initially I wanted to just make a few tweaks in the versions of example92.c, motor2.c and motor_test2.c above without actually having to understand the code. It turns out that one needs to carefully consider what various types and variables are used for. This can be a good thing or a bad thing depending on one's circumstances, goals and personality. The programs above evolved into example93.c, motor3.c and motor_test3.c. First we'll look at example93.c:

```
//////////////////////////////////////////////////////////////////
// example93.cpp
//
// Copyright (c) 2015 Robert Ramey
//
// Distributed under the Boost Software License, Version 1.0. (See
// accompanying file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

#include <iostream>

// include headers to support safe integers
#include <boost/safe_numerics/cpp.hpp>
#include <boost/safe_numerics/exception.hpp>
#include <boost/safe_numerics/safe_integer.hpp>
#include <boost/safe_numerics/safe_integer_range.hpp>
#include <boost/safe_numerics/safe_integer_literal.hpp>

// use same type promotion as used by the pic compiler
// target compiler XC8 supports:
using pic16_promotion = boost::safe_numerics::cpp<
    8,  // char      8 bits
    16, // short     16 bits
    16, // int       16 bits
    16, // long      16 bits
    32  // long long 32 bits
>;

// *************************
// 1. Specify exception policies so we will generate a
// compile time error whenever an operation MIGHT fail.

// *************************
// generate runtime errors if operation could fail
using exception_policy = boost::safe_numerics::default_exception_policy;

// generate compile time errors if operation could fail
using trap_policy = boost::safe_numerics::loose_trap_policy;
```

```
// ***************************
// 2. Create a macro named literal an integral value
// that can be evaluated at compile time.
#define literal(n) make_safe_literal(n, pic16_promotion, void)

// For min speed of 2 mm / sec (24.8 format)
// sec / step = sec / 2 mm * 2 mm / rotation * rotation / 200 steps
#define C0      literal(5000 << 8)

// For max speed of 400 mm / sec
// sec / step = sec / 400 mm * 2 mm / rotation * rotation / 200 steps
#define C_MIN   literal(25 << 8)

static_assert(
    C0 < make_safe_literal(0xffffff, pic16_promotion,trap_policy),
    "Largest step too long"
);
static_assert(
    C_MIN > make_safe_literal(0, pic16_promotion,trap_policy),
    "Smallest step must be greater than zero"
);

// ***************************
// 3. Create special ranged types for the motor program
// These wiil guarantee that values are in the expected
// ranges and permit compile time determination of when
// exceptional conditions might occur.

using pic_register_t = boost::safe_numerics::safe<
    uint8_t,
    pic16_promotion,
    trap_policy // use for compiling and running tests
>;

// note: the maximum value of step_t would be:
// 50000 = 500 mm / 2 mm/rotation * 200 steps/rotation.
// But in one expression the value of number of steps * 4 is
// used.  To prevent introduction of error, permit this
// type to hold the larger value.
using step_t = boost::safe_numerics::safe_unsigned_range<
    0,
    200000,
    pic16_promotion,
    exception_policy
>;

// position
using position_t = boost::safe_numerics::safe_unsigned_range<
    0,
    50000, // 500 mm / 2 mm/rotation * 200 steps/rotation
    pic16_promotion,
    exception_policy
>;

// next end of step timer value in format 24.8
// where the .8 is the number of bits in the fractional part.
using ccpr_t = boost::safe_numerics::safe<
    uint32_t,
    pic16_promotion,
    exception_policy
```

```
>;

// pulse length in format 24.8
// note: this value is constrainted to be a positive value. But
// we still need to make it a signed type. We get an arithmetic
// error when moving to a negative step number.
using c_t = boost::safe_numerics::safe_unsigned_range<
    C_MIN,
    C0,
    pic16_promotion,
    exception_policy
>;

// index into phase table
// note: The legal values are 0-3.  So why must this be a signed
// type?  Turns out that expressions like phase_ix + d
// will convert both operands to unsigned.  This in turn will
// create an exception.  So leave it signed even though the
// value is greater than zero.
using phase_ix_t = boost::safe_numerics::safe_signed_range<
    0,
    3,
    pic16_promotion,
    trap_policy
>;

// settings for control value output
using phase_t = boost::safe_numerics::safe<
    uint16_t,
    pic16_promotion,
    trap_policy
>;

// direction of rotation
using direction_t = boost::safe_numerics::safe_signed_range<
    -1,
    +1,
    pic16_promotion,
    trap_policy
>;

// some number of microseconds
using microseconds = boost::safe_numerics::safe<
    uint32_t,
    pic16_promotion,
    trap_policy
>;

// *************************
// emulate PIC features on the desktop

// filter out special keyword used only by XC8 compiler
#define __interrupt
// filter out XC8 enable/disable global interrupts
#define ei()
#define di()

// emulate PIC special registers
pic_register_t RCON;
pic_register_t INTCON;
```

```
pic_register_t CCP1IE;
pic_register_t CCP2IE;
pic_register_t PORTC;
pic_register_t TRISC;
pic_register_t T3CON;
pic_register_t T1CON;

pic_register_t CCPR2H;
pic_register_t CCPR2L;
pic_register_t CCPR1H;
pic_register_t CCPR1L;
pic_register_t CCP1CON;
pic_register_t CCP2CON;
pic_register_t TMR1H;
pic_register_t TMR1L;

// ***************************
// special checked type for bits - values restricted to 0 or 1
using safe_bit_t = boost::safe_numerics::safe_unsigned_range<
    0,
    1,
    pic16_promotion,
    trap_policy
>;

// create type used to map PIC bit names to
// correct bit in PIC register
template<typename T, std::int8_t N>
struct bit {
    T & m_word;
    constexpr explicit bit(T & rhs) :
        m_word(rhs)
    {}
    // special functions for assignment of literal
    constexpr bit & operator=(decltype(literal(1))){
        m_word |= literal(1 << N);
        return *this;
    }
    constexpr bit & operator=(decltype(literal(0))){
        m_word &= ~literal(1 << N);
        return *this;
    }
    // operator to convert to 0 or 1
    constexpr operator safe_bit_t () const {
        return m_word >> literal(N) & literal(1);
    }
};

// define bits for T1CON register
struct  {
    bit<pic_register_t, 7> RD16{T1CON};
    bit<pic_register_t, 5> T1CKPS1{T1CON};
    bit<pic_register_t, 4> T1CKPS0{T1CON};
    bit<pic_register_t, 3> T1OSCEN{T1CON};
    bit<pic_register_t, 2> T1SYNC{T1CON};
    bit<pic_register_t, 1> TMR1CS{T1CON};
    bit<pic_register_t, 0> TMR1ON{T1CON};
} T1CONbits;

// define bits for T1CON register
```

```
struct {
    bit<pic_register_t, 7> GEI{INTCON};
    bit<pic_register_t, 5> PEIE{INTCON};
    bit<pic_register_t, 4> TMR0IE{INTCON};
    bit<pic_register_t, 3> RBIE{INTCON};
    bit<pic_register_t, 2> TMR0IF{INTCON};
    bit<pic_register_t, 1> INT0IF{INTCON};
    bit<pic_register_t, 0> RBIF{INTCON};
} INTCONbits;

#include "motor3.c"

#include <chrono>
#include <thread>

// round 24.8 format to microseconds
microseconds to_microseconds(ccpr_t t){
    return (t + literal(128)) / literal(256);
}

using result_t = uint8_t;
const result_t success = 1;
const result_t fail = 0;

// move motor to the indicated target position in steps
result_t test(position_t new_position){
    try {
        std::cout << "move motor to " << new_position << '\n';
        motor_run(new_position);
        std::cout
        << "step #" << ' '
        << "delay(us)(24.8)" << ' '
        << "delay(us)" << ' '
        << "CCPR" << ' '
        << "motor position" << '\n';
        while(busy()){
            std::this_thread::sleep_for(std::chrono::microseconds(to_microseconds(c)));
            c_t last_c = c;
            ccpr_t last_ccpr = ccpr;
            isr_motor_step();
            std::cout << i << ' '
            << last_c << ' '
            << to_microseconds(last_c) << ' '
            << std::hex << last_ccpr << std::dec << ' '
            << motor_position << '\n';
        };
    }
    catch(const std::exception & e){
        std::cout << e.what() << '\n';
        return fail;
    }
    return success;
}

int main(){
    std::cout << "start test\n";
    result_t result = success;
    try {
        initialize();
        // move motor to position 1000
```

```
        result &= test(literal(1000));
        // move to the left before zero position
        // fails to compile !
        // result &= ! test(-10);
        // move motor to position 200
        result &= test(literal(200));
        // move motor to position 200 again! Should result in no movement.
        result &= test(literal(200));
        // move motor to position 50000.
        result &= test(literal(50000));
        // move motor back to position 0.
        result &= test(literal(0));
    }
    catch(...){
        std::cout << "test interrupted\n";
        return EXIT_FAILURE;
    }
    std::cout << "end test\n";
    return result == success ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Here are the changes we've made int the desktop test program

1. Specify exception policies so we can generate a compile time error whenever an operation MIGHT fail. We've aliased this policy with the name `trap_policy`. The default policy of which throws a runtime exception when an error is countered is aliased as `exception_policy`. When creating safe types, we'll now specify which type of checking, compile time or runtime, we want done.

2. Create a macro named "literal" an integral value that can be evaluated at compile time.

   "literal" values are instances of safe numeric types which are determined at compile time. They are `constexpr` values. When used along with other instances of safe numeric types, the compiler can calculate the range of the result and verify whether or not it can be contained in the result type. To create "literal" types we use the macro `make_safe_literal`(n, p, e) where n is the value, p is the promotion policy and e is the exception policy.

   When all the values in an expression are safe numeric values, the compiler can calculate the narrowest range of the result. If all the values in this range can be represented by the result type, then it can be guaranteed that an invalid result cannot be produced at runtime and no runtime checking is required.

   Make sure that all literal values are x are replaced with the macro invocation "literal(x)".

   It's unfortunate that the "literal" macro is required as it clutters the code. The good news is that is some future version of C++, expansion of `constexpr` facilities may result in elimination of this requirement.

3. Create special types for the motor program. These will guarantee that values are in the expected ranges and permit compile time determination of when exceptional conditions might occur. In this example we create a special type c_t to the width of the pulse applied to the motor. Engineering constraints (motor load inertia) limit this value to the range of C0 to C_MIN. So we create a type with those limits. By using limits no larger than necessary, we supply enough information for the compiler to determine that the result of a calculation cannot fall outside the range of the result type. So less runtime checking is required. In addition, we get extra verification at compile time that values are in reasonable ranges for the quantity being modeled.

   We call these types "strong types".

And we've made changes consistent with the above to motor3.c as well

```
/*
 * david austin
```

```
 * http://www.embedded.com/design/mcus-processors-and-socs/4006438/Generate-stepper-motor-speed-profiles-i
 * DECEMBER 30, 2004
 *
 * Demo program for stepper motor control with linear ramps
 * Hardware: PIC18F252, L6219
 *
 * Copyright (c) 2015 Robert Ramey
 *
 * Distributed under the Boost Software License, Version 1.0. (See
 * accompanying file LICENSE_1_0.txt or copy at
 * http://www.boost.org/LICENSE_1_0.txt)
 */

#include <assert.h>

// ramp state-machine states
enum ramp_state {
    ramp_idle = 0,
    ramp_up = 1,
    ramp_const = 2,
    ramp_down = 3,
};

// **************************
// 1. Define state variables using custom strong types

// initial setup
enum ramp_state ramp_sts;
position_t motor_position;
position_t m;              // target position
position_t m2;             // midpoint or point where acceleration changes
direction_t d;            // direction of traval -1 or +1

// curent state along travel
step_t i;                 // step number
c_t c;                    // 24.8 fixed point delay count increment
ccpr_t ccpr;              // 24.8 fixed point delay count
phase_ix_t phase_ix;     // motor phase index

// **************************
// 2. Surround all literal values with the "literal" keyword

// Config data to make CCP1&2 generate quadrature sequence on PHASE pins
// Action on CCP match: 8=set+irq; 9=clear+irq
phase_t const ccpPhase[] = {
    literal(0x909),
    literal(0x908),
    literal(0x808),
    literal(0x809)
}; // 00,01,11,10

void current_on(){/* code as needed */}  // motor drive current
void current_off(){/* code as needed */} // reduce to holding value

// **************************
// 3. Refactor code to make it easier to understand
// and relate to the documentation

bool busy(){
    return ramp_idle != ramp_sts;
```

```
}

// set outputs to energize motor coils
void update(ccpr_t ccpr, phase_ix_t phase_ix){
    // energize correct windings
    const phase_t phase = ccpPhase[phase_ix];
    CCP1CON = phase & literal(0xff); // set CCP action on next match
    CCP2CON = phase >> literal(8);
    // timer value at next CCP match
    CCPR1H = literal(0xff) & (ccpr >> literal(8));
    CCPR1L = literal(0xff) & ccpr;
}

// compiler-specific ISR declaration
// **************************
// 4. Rewrite interrupt handler in a way which mirrors the orginal
// description of the algorithm and minimizes usage of state variable,
// accumulated values, etc.
void __interrupt isr_motor_step(void) { // CCP1 match -> step pulse + IRQ
    // *** possible exception
    // motor_position += d;
    // use the following to avoid mixing exception policies which is an error
    if(d < 0)
        --motor_position;
    else
        ++motor_position;
    // *** possible exception
    ++i;
    // calculate next difference in time
    for(;;){
        switch (ramp_sts) {
            case ramp_up: // acceleration
                if (i == m2) {
                    ramp_sts = ramp_down;
                    continue;
                }
                // equation 13
                // *** possible negative overflow on update of c
                c -= literal(2) * c / (literal(4) * i + literal(1));
                if(c < C_MIN){
                    c = C_MIN;
                    ramp_sts = ramp_const;
                    // *** possible exception
                    m2 = m - i; // new inflection point
                    continue;
                }
                break;
            case ramp_const: // constant speed
                if(i > m2) {
                    ramp_sts = ramp_down;
                    continue;
                }
                break;
            case ramp_down: // deceleration
                if (i == m) {
                    ramp_sts = ramp_idle;
                    current_off(); // reduce motor current to holding value
                    CCP1IE = literal(0); // disable_interrupts(INT_CCP1);
                    return;
                }
```

```
                    // equation 14
                    // *** possible positive overflow on update of c
                    // note: re-arrange expression to avoid negative result
                    // from difference of two unsigned values
                    c += literal(2) * c / (literal(4) * (m - i) - literal(1));
                    if(c > C0){
                        c = C0;
                    }
                    break;
                default:
                    // should never arrive here!
                    assert(false);
            } // switch (ramp_sts)
            break;
    }
    assert(c <= C0 && c >= C_MIN);
    // *** possible exception
    ccpr = literal(0xffffff) & (ccpr + c);
    phase_ix = (phase_ix + d) & literal(3);
    update(ccpr, phase_ix);
} // isr_motor_step()

// set up to drive motor to pos_new (absolute step#)
void motor_run(position_t new_position) {
    if(new_position > motor_position){
        d = literal(1);
        // *** possible exception
        m = new_position - motor_position;
    }
    else
    if(motor_position > new_position){
        d = literal(-1);
        // *** possible exception
        m = motor_position - new_position;
    }
    else{
        d = literal(0);
        m = literal(0);
        ramp_sts = ramp_idle; // start ramp state-machine
        return;
    }

    i = literal(0);
    m2 = m / literal(2);

    ramp_sts = ramp_up; // start ramp state-machine

    T1CONbits.TMR1ON = literal(0); // stop timer1;

    current_on(); // current in motor windings

    c = C0;
    ccpr = (TMR1H << literal(8) | TMR1L) + C0 + literal(1000);
    phase_ix = d & literal(3);
    update(ccpr, phase_ix);

    CCP1IE = literal(1); // enable_interrupts(INT_CCP1);
    T1CONbits.TMR1ON = literal(1); // restart timer1;
} // motor_run()
```

```
void initialize() {
    di();                  // disable_interrupts(GLOBAL);
    motor_position = literal(0);
    CCP1IE = literal(0); // disable_interrupts(INT_CCP1);
    CCP2IE = literal(0); // disable_interrupts(INT_CCP2);
    PORTC = literal(0);  // output_c(0);
    TRISC = literal(0);  // set_tris_c(0);
    T3CON = literal(0);
    T1CON = literal(0x35);
    INTCONbits.PEIE = literal(1);
    INTCONbits.RBIF = literal(0);
    ei();                  // enable_interrupts(GLOBAL);
} // initialize()
```

1. Define variables using strong types

2. Surround all literal values with the "literal" keyword

3. Re-factor code to make it easier to understand and compare with the algorithm as described in the original article.

4. Rewrite interrupt handler in a way which mirrors the original description of the algorithm and minimizes usage of state variable, accumulated values, etc.

5. Distinguish all the statements which might invoke a runtime exception with a comment. There are 12 such instances.

Finally we make a couple minor changes in motor_test3.c to verify that we can compile the exact same version of motor3.c on the PIC as well as on the desktop.

Summary

The intent of this case study is to show that the Safe Numerics Library can be an essential tool in validating the correctness of C/C++ programs in all environments - including the most restricted.

- We started with a program written for a tiny micro controller for controlling the acceleration and deceleration of a stepper motor. The algorithm for doing this is very non-trivial and difficult prove that it is correct.

- We used the type promotion policies of the Safe Numerics Library to test and validate this algorithm on the desk top. The tested code is also compiled for the target micro controller.

- We used *strong typing* features of Safe Numerics to check that all types hold the values expected and invoke no invalid implicit conversions. Again the tested code is compiled for the target processor.

What we failed to do is to create a version of the program which uses the type system to prove that no results can be invalid. I turns out that states such as

```
++i;
c = f(c);
```

can't be proved not to overflow with this system. So we're left with having to depend upon exhaustive testing. It's not what we hoped, but it's the best we can do.

5. 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, from my perspective it 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.

Using later versions of C++ and the its standard library, template metaprogramming and Boost libraries I managed to (re)implement similar functionality in under 2000 ? lines of code. I promoted this version as a possible submission to the Boost. The feedback I received convinced me that no such library would be considered acceptable to the large majority of C++ programmers. It seems that the desire for maximum performance overrides any requirement that a program be known to be free of bugs. By this time I had a better idea of the opportunities available with the latest version of C++ (C++14) and resolved to address this issue by creating a library which would provide all the facilities of safe numerics at minimal runtime cost. The result is what you see here. The library now consists of 7000 lines of code, approximately 50 separate tests and more than 60 pages of documentation and examples.

Since I wrote the above, I've been contacted by David LeBlanc. He's been updating his package and informs me that the latest version:

- SafeInt does not require porting for different compilers, is fully supported on gcc, clang, and Visual Studio.

- The license has been changed from MS-PL to MIT license.

- The library has had a test suite since before it was public, and is now located here:

- SafeInt also has no external dependencies other than standard library files, and doesn't need anything else installed to work.

His current package can now be found at in github.

6. Type Requirements

6.1. 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
`t, u`	An object of a type modeling Numeric

Associated Types

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

Valid Expressions

In addition to the expressions defined in Assignable 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 Type	Return Value
`std::numeric_limits<T>::is_bounded`	`bool`	`true` or `false`
`std::numeric_limits<T>::is_integer`	`bool`	`true` or `false`
`std::numeric_limits<T>::is_signed`	`bool`	`true` or `false`
`std::numeric_limits<T>::is_specialized`	`bool`	`true`

Table 1.2. Unary Operators

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

Table 1.3. Binary Operators

Expression	Return Type	Semantics
`t - u`	`V`	subtract u from t
`t + u`	`V`	add u to t
`t * u`	`V`	multiply t by u
`t / u`	`T`	divide t by u

Expression	Return Type	Semantics
`t % u`	`T`	t modulus u
`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`	`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

Models

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

Header

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

Note on Usage of `std::numeric_limits`

We define the word "Numeric" in terms of the operations which are supported by "Numeric" types. This is in line with the current and historical usage of the word "concept" in the context of C++. It is also common to define compile time predicates such as "`is_numeric<T>`" to permit one to include expressions in his code which will generated a compile time error if the specified type (T) does not support the operations required. But this is not always true. In the C++ standard library there is a predicate `is_arithmetic<T>` whose name might suggest that it should return `true` for any type which supports the operations above. But this is not the case. The standard defines `is_arithmetic<T>` as `true` for any of the builtin types `int`, `long`, `float`, `double`, etc and `false` for any other types. So even if a user defined type U were to support the operations above, `is_arithmetic<U>` would still return `false`. This is quite unintuitive and not a good match for our purposes. Hence we define our own term "Numeric" to designate any type T which:

- Supports the operations above

- Specializes the standard type numeric_limits

while following the C++ standard in using `is_arithmetic<T>`, `is_integral<T>` to detect specific types only. The standard types are useful in various aspects of the implementation - which of course is done in terms of the standard types.

This in turn raises another question: Is it "legal" to specialize `std::numeric_limits` for one's own types such as `safe<int>`. In my view the standard is ambiguous on this. See various interpretations:

- is-it-ok-to-specialize-stdnumeric-limitst-for-user-defined-number-like-class

- cpppreference.com/w/cpp/types/numeric_limits

In any case, it seems pretty clear that no harm will come of it. In spite of the consideration given to this issue, it turns out that the found no real need to implement these predicates. For example, there is no "is_numeric<T>" implemented as part of the safe numerics library. This may change in the future though. Even if not used, defining and maintaining these type requirements in this document has been very valuable in keeping the concepts and code more unified and understandable.

Remember that above considerations apply to other numeric types used in this library even though we don't explicitly repeat this information for every case.

6.2. 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

`T, U, V`	A type that is a model of the Integer
`t, u`	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
`std::numeric_limits<T>::is_integer`	true

Expression	Return Type	Semantics
`~t`	T	bitwise complement
`t << u`	T	shift t left u bits

Expression	Return Type	Semantics
`t >> u`	`T`	shift t right by u bits
`t & u`	`V`	and of t and u padded out to max # bits in t, u
`t	u`	`V`	or of t and u padded out to max # bits in t, u
`t ^ u`	`V`	exclusive or of t and u padded out to max # bits in t, u
`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

Models

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

Header

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

6.3. SafeNumeric<T>

Description

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

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

- 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 the 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`.

Invariants

The fundamental requirement of a SafeNumeric type is that it implements all C++ operations permitted on its 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..). 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>`

6.4. PromotionPolicy<PP>

Description

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

```
int x;
char y;
```

```
auto z = x + y
```

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

Notation

PP	A type that full fills the requirements of a PromotionPollicy
T, U	A type that is a model of the Numeric concept
R	An object of type modeling Numeric which can be used to construct a SafeNumeric 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 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 he result of x + y. In this case that will be a long int (or perhaps a long long) depending upon the compiler and machine architecture. In this case, there will be no need for any special checking on the result and there can be no overflow.

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

  This policy is documented in Promotion Policies - automatic

- boost::numeric::cpp

  Use expression type promotion rules to emulate another processor. When this policy is used, C++ type for safe integers follows the rules that a compiler on the target processor would use. This permits one to test code destined for a one processor on the another one. One situation there this can be very, very useful is when testing code destined for a micro controller which doesn't have the logging, debugging, input/output facilities of a desktop.

```
// specify a promotion policy to support proper emulation of
// PIC 18f2520 types on the desktop
using pic16_promotion = boost::numeric::cpp<
    8,  // char      8 bits
    16, // short     16 bits
    16, // int       16 bits
    16, // long      16 bits
    32  // long long 32 bits
>;
...
safe<std::uint16_t, pic16_promotion> x, y;
...

x + y; // detect possible overflow on the pic.
```

For a complete example see Safety Critical Embedded Controller.

Header

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

6.5. 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 ExceptionPolicy
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 yield an invalid result, one of the following functions will be invoked.

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

7. Types

7.1. 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.

Model of

Integer

SafeNumeric

This type inherits all the notation, associated types and template parameters and valid expressions of SafeNumeric types. The following specify additional features of this type.

Notation

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

Associated Types

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

Template Parameters

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

See examples below.

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 of 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 result. 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::safe_numerics;

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::safe_numerics;

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 <boost/safe_numerics/safe_integer.hpp>
#include <boost/safe_numerics/safe_integer_literal.hpp>

using namespace boost::safe_numerics;

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 promotion policy `automatic`. The policy stores the result of an expression in the smallest size type that can accommodate the largest value that an expression can yield. No checking for exceptions is necessary. The following example illustrates this.

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

int main(int, char[]){
    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

    return 0;
}
```

Header

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

7.2. 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
MIN, MAX	Minimum and maximum values that the range can represent.

Associated Types

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

Template Parameters

Parameter	Requirements	Description
MIN	must be 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 `boost::numeric::native`
EP	Exception Policy<EP>	Default value is `boost::numeric::default_exception_policy`

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 <boost/safe_numerics/safe_integer.hpp>
#include <boost/safe_numerics/safe_integer_range.hpp>

#include <boost/safe_numerics/utility.hpp>

using namespace boost::safe_numerics;

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]"
    );
}

int main(){}
```

Header

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

7.3. 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.

Model of

Integer

SafeNumeric

This type inherits all the notation, associated types and template parameters and valid expressions of SafeNumeric types. The following specify additional features of this type.

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 void
EP	Exception Policy<EP>	Default value is void

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. The 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 because there are
// no promotion and exception policies specified.
constexpr safe<int> z = x / y;
```

Valid Expression

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

Example of use

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

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

make_safe_literal(n, PP, EP)

This is a macro which returns an instance of a safe literal type. This instance will hold the value n. The type of the value returned will be the smallest safe type which can hold the value n.

Header

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

7.4. 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 that 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 so it's not an exact match, but it's useful and works 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 it's 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.
domain_error	the result of an operation is outside the legal range of the result.
range_error	an argument to a function or operator is outside the legal range - e.g. sqrt(-1).
precision_overflow_error	precision was lost in the course of executing the operation.
underflow_error	A number is too close to zero to be represented by the data type.
uninitialized_value	According to the C++ standard, the result may be defined by the application. e.g. 16 >> 10 will result the expected result of 0 on most machines.

The above listed codes can be transformed to a 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>
```

7.5. 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 AE(e, message)	The operation cannot produce valid arithmetic result such as overflows, divide by zero, etc.

Parameter	Type Requirements	Invoked when:
UB	Function object callable with the expression UB(e, message)	The result is undefined by the C++ standard
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 std::system_error
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
loose_exception_policy	Throws runtime exception on any arithmetic error. Undefined and implementation defined behavior is permitted as long as it does not produce an arithmetic error.
loose_trap_policy	Invoke a compile time error in any case where it's possible to result in an arithmetic error.
strict_exception_policy	Throws runtime exception on any arithmetic error. Any undefined or implementation defined behavior also results in throwing an exception.
strict_trap_policy	Invoke a compile time error in any case where it's possible to result in an arithmetic error, undefined behavior or implementation defined behavior
default_exception_policy	an alias for strict_exception_policy

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>
```

7.6. 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 guarentee
    // that there will be no incorrect result.
    // In such cases, there is no runtime overhead from using safe types.
    safe_int8 x = 127;
    safe_int8 y = 2;
    safe<int, native> z; // z can now hold the result of the addition of any two 8 bit numbers
    z = x + y; // is 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 operants, 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. For an extended example on how this is done, look at Safety Critical Embedded Controller .

Header

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

8. Exception Safety

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

9. 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 and likely useful in their own right. For all these reasons, they are documented here.

In general terms, the library works in the following manner:

At compile time:

- The library defines "safe" versions of C++ primitive arithmetic types such as `int`, `unsigned int`, etc.

- Arithmetic operators are defined for these "safe" types. These operators are enhanced versions of the standard C/C++ implementations. These operators are declared and implemented in the files "safe_base.hpp" and "safe_base_operations.hpp".

- For binary operators, verify that both operands have the same promotion and and exception handling policies. If they don't, invoke compilation error.

- Invoke the promotion policy to determine the result type R of the operation.

- For each operand of type T retrieve the range of values from `std::numeric_limits<T>::min()` and `std::numeric_limits<T>::max()`. A range is a pair of values representing a closed interval with a minimum and maximum value.

- These ranges are cast to equivalent values of the result type, R. It's possible that values cannot be cast to the result type so the result of the cast is returned as a variant type, `checked_result<R>. checked_result<R>` may hold either a value of type

R or a `safe_numerics_error` value indicating why the cast could not be accomplished. Ranges are represented as a pair of values of the type `checked_result<R>`.

- `checked_result<R>` can be considered enhanced versions of the underlying type R. Operations which are legal on values of type R such as +, -, ... are also legal on values of `checked_result<R>`. The difference is that the latter can record operation failures and propagate such failures to subsequent operations. `checked_result<R>` is implemented in the header file "checked_result.hpp". Operations on such types are implemented in "checked_result_operations.hpp".

- Given the ranges of the operands, determine the range of the result of the operation using compile-time interval arithmetic. The `constexpr` facility of C++14 permits the range of the result to be calculated at compile time. Interval arithmetic is implemented in the header file "interval.hpp". The range of the result is also represented as a pair of values of the type `checked_result<R>`.

- Operations on primitives are implemented via free standing functions described as checked arithmetic. These operations will return instances of `checked_result<R>` .

At run time:

- If the range of the result type includes only arithmetically valid values, the operation is guaranteed to produce an arithmetically correct result and no runtime checking is necessary. The operation invokes the original built-in C/C++ operation and returns the result value.

- Otherwise, operands are cast to the result type, R, according to the selected promotion policy. These "checked" cast operations return values of type `checked_result<R>`.

- If either of the casting operations fails, an exception is handled in accordance with the exception policy.

- Otherwise, the operation is performed using "checked arithmetic". These free functions mirror the normal operators +, -, *, ... except that rather than returning values of type R, they return values of the type `checked_result<R>`. They are defined in files "checked_default.hpp", "checked_integer.hpp" ,"checked_float.hpp".

- If the operation is not successful, the designated exception policy function is invoked.

- Otherwise, the result value is returned as a `safe<R>` type with the above calculated result range.

The following components realize the design described here.

9.1. checked_result<R>

Description

checked_result is a special kind of variant 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. It is similar to other types proposed for and/or included to the C++ standard library or Boost such as `expected`, `variant`, `optional` and `outcome`. In some circumstances it may be referred to as a "monad".

- All instances of `checked_result<R>` are immutable. That is, once constructed, they cannot be altered.

- There is no default constructor.

- `checked_result<R>` is never empty.

- Binary operations supported by type R are guaranteed to be supported by checked_result<R>.

- Binary operations can be invoked on a pair of `checked_result<R>` instances if and only if the underlying type (R) is identical for both instances. They will return a value of type `checked_result<R>`.

- Unary operations can be invoked on `checked_result<R>` instances. They will return a value of type `checked_result<R>`.

- Comparison operations will return a `boost::logic::tribool`. Other binary operations will a value of the same type as the arguments.

Think of `checked<R>` as an "extended" version of R which can hold all the values that R can hold in addition other "special values". For example, consider checked<int>.

Notation

Symbol	Description
R	Underlying type
r	An instance of type R
c, c1, c2	an instance of checked_result<R>
t	an instance of checked_result<T> for some type T not necessarily the same as R
e	An instance of type `safe_numerics_error`
msg	An instance of type const char *
OS	A type convertible to `std::basic_ostream`
os	An instance of type convertible to `std::basic_ostream`

Template Parameters

R must model the type requirements of Numeric

Parameter	Description
R	Underlying type

Model of

Numeric

Valid Expressions

All expressions are `constexpr`.

Expression	Return Type	Semantics
checked_result(r)	checked_result<R>	constructor with valid instance of R

Expression	Return Type	Semantics
`checked_result<R>(t)`	`checked_result<R>`	constructor with `checked_result<T>` where T is not R. T must be convertible to R.
`checked_result(e, msg)`	`checked_result<R>`	constructor with error information
`static_cast<R>(c)`	R	extract wrapped value - compile time error if not possible
`static_cast<safe_numerics_error>(c)`	`safe_numerics_error`	extract wrapped value - may return `safe_numerics_error::success` if there is no error
`static_cast<const char *>(c)`	`const char *`	returns pointer to the included error message
`c.exception()`	`bool`	true if `checked_result` contains an error condition.

`c1 < c2`
`c1 >= c2`
`c1 > c2`
`c1 <= c2`
`c1 == c2`
`c1 != c2`
	`boost::logic::tribool`	compare the wrapped values of two checked_result instances. If the values are such that the result of such a comparison cannot be reasonably defined, The result of the comparison is `boost::logic::tribool::indeterminant`.

`c1 + c2`
`c1 - c2`
`c1 * c2`
`c1 / c2`
`c1 % c2`
`c1 | c2`
`c1 & c2`
`c1 ^ c2`
`c1 << c2`
`c1 >> c2`
	`checked_result<R>`	returns a new instance of `checked_result<R>`.

`os << c`	OS	writes result to output stream. If the result is an error it writes the string corresponding to the error message. Otherwise, it writes the numeric value resulting from the operation. Returns reference to output stream.

Example of use

```
//   Copyright (c) 2018 Robert Ramey
//
// Distributed under the Boost Software License, Version 1.0. (See
// accompanying file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

#include <iostream>

#include <boost/safe_numerics/checked_result.hpp>
#include <boost/safe_numerics/checked_result_operations.hpp>

int main(){
    using ext_uint = boost::safe_numerics::checked_result<unsigned int>;
    const ext_uint x{4};
```

```
    const ext_uint y{3};

    // operation is a success!
    std::cout << "success! x - y = " << x - y;

    // subtraction would result in -1, and invalid result for an unsigned value
    std::cout << "problem: y - x = " << y - x;

    const ext_uint z = y - x;
    std::cout << "z = " << z;
    // sum of two negative overflows is a negative overflow.
    std::cout << "z + z" << z + z;

    return 0;
}
```

See Also

ExceptionPolicy

Header

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

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

9.2. Checked Arithmetic

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 Numeric type requirements.

Complexity

Each function performs one and only one arithmetic operation.

Example of use

```
#include <boost/numeric/safe_numerics/checked_default.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.

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>
checked_result<R> constexpr checked::add(const R & t, const R & u);

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

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

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

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

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

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

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

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

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

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

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

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

template<class R>
```

```
checked_result<R> constexpr checked::bitwise_and(const R & t, const R & u);

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

See Also

checked_result<R>

Header

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

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

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

9.3. interval<R>

Description

A closed arithmetic interval represented by a pair of elements of type R. In principle, one should be able to use Boost.Interval library for this. But the functions in this library are not `constexpr`. Also, this Boost.Interval is more complex and does not support certain operations such bit operations. Perhaps some time in the future, Boost.Interval will be used instead of this `interval<R>` type.

Template Parameters

R must model the type requirements of Numeric. Note this in principle includes any numeric type including floating point numbers and instances of `checked_result<R>`.

Notation

Symbol	Description
I	An interval type
i, j	An instance of interval type
R	Numeric types which can be used to make an interval
r	An instance of type R
p	An instance of std::pair<R, R>
l, u	Lowermost and uppermost values in an interval
os	std::basic_ostream<class CharT, class Traits = std::char_traits<CharT>>

Associated Types

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

Valid Expressions

Note that all expressions are constexpr.

Expression	Return Type	Semantics
`interval<R>(l, u)`	`interval<R>`	construct a new interval from a pair of limits
`interval<R>(p)`	`interval<R>`	construct a new interval from a pair of limits
`interval<R>(i)`	`interval<R>`	copy constructor
`make_interval<R>()`	`interval<R>`	return new interval with std::numric_limits<R>::min() and std::numric_limits<R>::max()
`make_interval<R>(const R &r)`	`interval<R>`	return new interval with std::numric_limits<R>::min() and std::numric_limits<R>::max()
`i.l`	`R`	lowermost value in the interval i
`i.u`	`R`	uppermost value in the interval i
`i.includes(j)`	`boost::logic::tribool`	return true if interval i includes interval j
`i.excludes(j)`	`boost::logic::tribool`	return true if interval i includes interval j
`i.includes(t)`	`bool`	return true if interval i includes value t
`i.excludes(t)`	`bool`	return true if interval i includes value t
`i + j`	`interval<R>`	add two intervals and return the result
`i - j`	`interval<R>`	subtract two intervals and return the result
`i * j`	`interval<R>`	multiply two intervals and return the result
`i / j`	`interval<R>`	divide one interval by another and return the result
`i % j`	`interval<R>`	calculate modulus of one interval by another and return the result
`i << j`	`interval<R>`	calculate the range that would result from shifting one interval by another
`i >> j`	`interval<R>`	calculate the range that would result from shifting one interval by another
`i | j`	`interval<R>`	range of values which can result from applying | to any pair of operands from I and j
`i & j`	`interval<R>`	range of values which can result from applying & to any pair of operands from I and j
`i ^ j`	`interval<R>`	range of values which can result from applying ^ to any pair of operands from I and j

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

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>
```

9.4. 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 <, >, >=, <=, ==, != and shouldn't be used by user programs which should be portable to standard C/C++ integer arithmetic. 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>
```

10. 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
```

11. Rationale and FAQ

11.1. 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. I believe they convincingly demonstrate that any program which does not use this library must be assumed to contain arithmetic errors.

11.2. 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.

11.3. 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.

11.4. 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].

11.5. 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 primitive values at compile time.

11.6. 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.

11.7. Why is Boost.Convert not used?

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

11.8. 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.

11.9. 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.

11.10. 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]

11.11. 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 any rigorous way.

11.12. 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 common 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].

11.13. 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.

11.14. 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.

12. Pending Issues

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

12.1. `safe_base` Only Works for Scalar Types

The following is paraphrased from an issue raised by Andrzej Krzemie#ski as a github issue. It touches upon fundamental ideas behind the library and how these ideas as the implementation of the library collided with reality.

"In the current implementation safe<T> will only work with T being a C++ scalar type. Therefore making a general type requirements that say what operations are allowed is superfluous, and confusing (because it implies that safe<> is more generic."

When I started out, It became clear that I wanted "safe" types to look like "numeric" types. It also became clear pretty soon that there was going to be significant template meta-programming in the implementation. Normal type traits like std::is_integer are defined in the std namespace and one is discouraged from extending it. Also I needed some compile time "max" and "lowest" values. This lead me to base the design on std::numeric_limits. But std::numeric limits is inherently extensible to any "numeric" type. For example, money is a numeric type but not an intrinsic types. So it seemed that I needed to define a "numeric" concept which required that there be an implementation of std::numeric_limits for any type T - such as money in this case. When I'm doubt - I tend to think big.

For now though I'm not going to address it. For what it's worth, my preference would be to do something like:

```
template<typename T>
struct range {
    T m_lowest;
    T m_highest;
    // default implementation
    range(
        const & T t_min,
        const & T t_max
    ) :
        m_lowest(std::numeric_limits<T>::lowest(t_min),
        m_highest(std::numeric_limits<T>::max(t_max)
    {}
};
```

Then redeclare `safe_base`, etc., accordingly.

Also not that for C++20, template value parameters are no longer restricted to integer primitive types buy maybe class types as well. This means the library maybe extended to user class types without changing the current template signatures.

12.2. Concepts are Defined but Not Enforced.

The following is paraphrased from an issue raised by Andrzej Krzemie#ski as a github issue.

"You do not need a concept to constrain anything with it, in your library. Or is the purpose of the Type requirements to show in detail what it means that safe<T> is a 'drop-in replacement for T?"

Right - currently I don't use the concept to constrain anything. They are currently a purely "conceptual" tool to keep the design from getting off track. This is common with other libraries such as the C++ standard library where the concepts are defined but not enforced by compile time predicates. Hopefully in future that might change - see below

"If you want to extend safe<T> for other integer types, Type requirement still need to be fixed:"

Hmmmm - I'm not quite sure that this is true. One thing that IS true is the the interface and implementation of the library will need to be enhanced to permit "safe" to be applied to user defined types. This is apparent now, but as my brain can only comprehend the library one piece at a time, this design feature was lost during the implementation. In implementing co-existence of floats with safe integers, I did refactor the implementation in a way which I believe my eventually permit the application to any user supplied T which implements all the required operations of Numeric types. So as it is now this is pending. If the library were

to become widely used, there might be motivation to do this. Time will tell. So for now I'm leaving these in the documentation and code, even though they are not actually used.

12.3. `Safe` values Can Be Uninitialized

Currently the library permits a `safe<int>` value to be uninitialized. This supports the goal of "drop-in" replacement of C++/C built-in types with safe counter parts. On the other hand, this breaks the "always valid" guarantee. The recent improvements in the exception policy implementation will permit a user to specify how this is to be handled.

12.4. Other Pending Issues

• The library is currently limited to integers. If there is interest, it could be extended to floats and possible to user defined types.

• Although care has been taken to make the library portable, at least some parts of the implementation - particularly `checked` integer arithmetic - depend upon two's complement representation of integers. Hence the library is probably not currently portable to all other possible C++ architectures. These days, this is unlikely to be a limitation in practice. Starting with C++20, integer arithmetic will be guaranteed by the C++ standard to be two's complement.

• `std::common_type` is used in a variety of generic libraries, including std::chrono. Without a specialization for `safe<T>`s one cannot use the safe wrappers e.g. as a representation for `std::chrono::duration`.

13. 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

14. Release Log

This is the third version.

Revision History
Revision 1.69   29 September 2018
First Boost Release
Revision 1.70   9 March 2019
Fixed Exception Policies for `trap` and `ignore`.

15. 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 - ISOCPP . January 15, 2012.

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

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

[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.

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

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

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

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

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

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

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

[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.

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

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

[INT32-C] 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.

[Stone] David Stone. *C++ Bounded Integer Library* .

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

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