Safe Arithmetic User Guide

Motivation

C++ represents numerical values with finite bounds and discrete steps, as do all commonly used programming languages and processors. This is not a problem as long as specific rules regarding overflow, underflow, and loss of precision are meticulously followed. There are a number of guidelines and standards published for functional safety and security that detail the rules for safe arithmetic operations as well as many other concerns:

- MISRA C and MISRA C++ Guidelines
- CERT C and CERT C++ Secure Coding Standards
- ISO/IEC TS 17961:2013 (C Secure Coding Rules)
- IEC 61508 (Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems)

These publications represent a large investment in the safe application of software, and for good reason. The consequences for such bugs can be catastrophic.

- Therac-25 radiotherapy machine. An integer overflow caused the machine to bypass software safety checks and deliver deadly radiotherapy doses to patients.
- Ariane rocket flight V88. An integer overflow was caught, but lead to an exception that was handled incorrectly, thus halting its inertial navigation system. This ultimately led to the catastrophic destruction of the rocket and its payload.
- Boeing 787 computer crash. Two different integer overflows caused either misleading data to be displayed or systems crash.

There are countless other functional bugs and security vulnerabilities all due to unsafe arithmetic operations.

The Safe Arithmetic library is intended and designed to protect against unsafe arithmetic operations. This includes overflow, underflow, and divide-by-zero. It follows the "Valley of success" strategy; The natural state of a software project utilizing the Safe Arithmetic library contains clear boundaries between safe and unsafe arithmetic. In this context, arithmetic includes assignment and construction.

```
TODO: describe support for integer, fixed point, flow, arbitrary precision, pointers
```

```
TODO: list other libraries that intend to solve the same or similar problem
```

Overview

Safe Arithmetic Environment

The Safe Arithmetic library uses the C++ type system to encode and enforce requirements on values. A special template type, `safe::var` is used to contain these values.

```
namespaced safe {
    template<typename ValueType, auto Requirements>
    struct var;
}
```

Arithmetic, bitwise, and shift operators on `safe::var` values results in the generation of a new `safe::var` with its requirements updated to represent the set of possible values the result may contain. Operations on `safe::var` values are guaranteed to be safe at compile-time. There is no runtime overhead incurred. Only the desired operations are performed.

Operations on instances of `safe::var` forms a hermetically sealed context in which overflows, underflows, and division-by-zero are proven impossible by the Safe Arithmetic library implementation. If such a condition were possible due to an arithmetic operation on a `safe::var`, then compilation would fail.

This leaves two important questions, how to get values in and out of this "Safe Arithmetic Environment".

Creating Safe Values

```
namespace safe::literals {
    template <char... chars>
    constexpr auto operator""_i();
}
```

```
namespace safe {
    // safe versions of C++ primitive integer types
    using u8 = var<std::uint8_t, ...>;
    using s8 = var<std::int8_t, ...>;
    using u16 = var<std::uint16_t, ...>;
    using s16 = var<std::int16_t, ...>;
    using u32 = var<std::uint32_t, ...>;
    using s32 = var<std::int32_t, ...>;
    using u64 = var<std::uint64_t, ...>;
    using s64 = var<std::int64_t, ...>;
}
```

```
namespace safe {
    // safe arbitrary precision signed integer type
    template<std::size_t NumBits>
    using integer = var<...>;
}
```

```
namespace safe {
    template<typename Return>
    Return match<Return>(auto... matches, auto no_match);
}
```

The final method of introducing values into the safe arithmetic environment is through `unsafe_cast<T>(value)`. It bypasses all the compile-time and runtime safety checks and depends on the value to be proven to satisfy the requirements of `T` using mechanisms outside the visibility of the Safe Arithmetic library. It should almost never be used. Its use is highly discouraged. The name was chosen to cause an uneasy feeling in programmers and blare out a red-alert alarm for code reviewers.

```
template<typename T>
T unsafe_cast(auto value);
```

Extracting Safe Values

```
template<typename T>
T safe_cast(auto value);
```