MBDA

ADDING ROBUSTNESS TO C++

JOHN PRICE, MBDA, TECHNICAL EXPERT SOFTWARE DESIGN AND IMPLEMENTATION

HISC

HIGH INTEGRITY SOFTWARE CONFERENCE

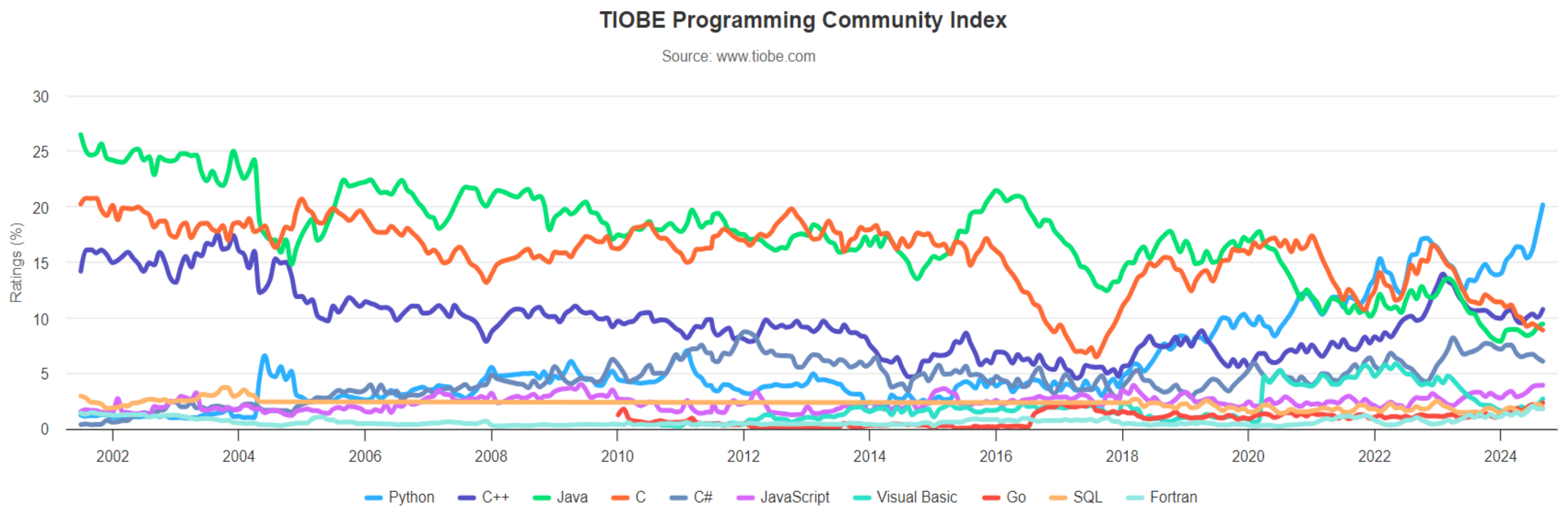
OCT 22, 2024

Table of contents

Introduction

Background

- MBDA heavy Ada user however…
- Ada has small market share
- Socio-economic pressures
- Company policy now rejects the use of Ada for new projects
- Looking at Rust but…



Language	Position	Rating
C++	2	10.75%
C	4	8.89%
Rust	14	1.32%
Ada	26	0.71%

Reference

Desired Language features

- Strong ranged types
    - Integers
    - Floats
    - Enums
    - Fixed point types
    - Modulo types
- Non zero based or enum based Array indexing
- Ability to represent any arbitrary data structure
- Variant records
- Compile and runtime checks
- Software exceptions with output of stack trace

The Challenge

- Can we represent these features in C++?
- Can we force enough code to run at compile time to conduct the compile time checks?
- Will the user applications written using the library run at an acceptable speed?

Things we wanted to avoid

- Dynamic memory allocation
- Macros
- Obscure error messages
- Syntactic gymnastics

Existing libraries

- Boost "safe" library defines templates for use with basic types
  - provides overflow checks etc.
  - does not allow further range restriction
- Other libraries exist for ranged types
  - these provide a useful reference
  - do not provide a comprehensive set of features

Compiler Issues

- Having used gcc and Visual Studio, what C++ will compile seems to be very compiler dependent….
- The Strong library relies on C++17 template features. However,
  - GCC compiler bug 85282 prevents template specialization inside other templates
  - Had to move from template specialization to constexpr if. However Visual Studio not great with constexpr if…
- Looking to move to C++ 20 to take advantage of “concepts”
  - This allows the compiler to check the validity of the template parameters
- C++ 23 may introduce “reflection” allowing built in enum to string conversions

Basic Principals

Range Definition

- A simple ranged type definition might be

```
using distance_type = strong::type<uint8_t,1,7>;
using velocity_type = strong::type<uint8_t,1,7>;
```

- However, C++ cannot pass float values as a template parameter
- We need to differentiate between types with the same min/ max values

- Defining a range template provides the key.

```
constexpr strong::range<uint8_t> distance_range (1,7);
using distance_type = strong::type<distance_range>;

constexpr strong::range<uint8_t> velocity_range (1,7);
using velocity_type = strong::type<velocity_range>;
```

- Assigning an attribute of velocity_type to distance_type will cause a compilation error
- If an attribute of velocity_type is set to 8 the software is directed to a last chance handler. This is can be overridden but the default behaviour is to generate a stack trace and terminate the software.

Declaration in headers

- The declarations used so far work fine in an implementation file
- However, in a header used by say 3 implementation files this would result in 3 incompatible types. Each would have a different instance of the range variable.
- Range needs "static linkage"

```
struct def_radian_type {
        static constexpr strong::range<double> radian_type_range(-3.14,3.14);
}
using radian_type = strong::type<def_radian_type::radian_type_range>;
```

- This is starting to get too much for general use
- So we have a macro.

```
STRONG_TYPE(radian_type, double, -3.14,3.14);
```

- And for enums

```
STRONG_ENUM(fish_type, cod, haddock, plaice, sole);
```

- This enables us to include enum to string conversion functions.
  - Until we get "reflection" in C++23

Other Types

- Fixed point types are provided, in the range definition the parameters are treated as doubles. The LSB is provided as the final parameter.

```
STRONG_FIXED_TYPE(location_type,int8_t,-30,30,0.5);
location_type location = 20.5;
```

- Modular types are provided

```
STRONG_MODULO_TYPE(modulo_8_type,int8_t,0,7);
modulo_8_type modulo_8 = 4;
modulo_8 = modulo_8 + 5;        // molulo_8 = 1
```

- In order to conduct fixed point maths without using floating point operations the strong library uses a class called universal_fixed. This represents fractional numbers using a bool for sign and uint64_t for integral, numerator and denominator.

```
strong::universal_fixed foo = 1.5;        // true, 1,1,2
strong::universal_fixed baa = -0.2;       // false,0,1,5
strong::universal_fixed result = foo*baa; // false,0,3,10
double result2 = result.get_value();      // -0.3
```

- Limited types are supported

```
STRONG_LIMITED(limited_degree_type,uint16_t,0,360);
```

Reference

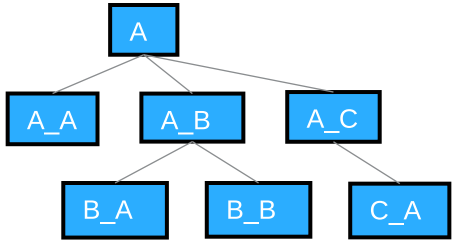
Subtypes

Subtype hierarchy

- Subtypes are compatible types with a constrained range

```
        STRONG_TYPE(degree_type,float,-180f,180f);
        STRONG_TYPE(semi_circle_type,degree_type,0f,180f);
```

- Variables of degree_type and semi_circle_type may be mixed in mathematical expressions without explicit conversion
- Subtypes can be used to create an arbitrary tree of sub-types



- In the strong library each child is considered to be of the same type as its parent but not its siblings. Therefore C_A can be assigned to A but B_B can only be assigned to B_A via type A_B.

Fixed Point Subtypes

- Fixed subtypes may be generated using the strong type macro

```
STRONG_FIXED_TYPE(fixed_radian_type,int32_t,-3.14f,3.14f,0.01f);
STRONG_TYPE(fixed_semi_type,fixed_radian_type,0.0f,3.14f);
```

Type conversion

- For explicit type conversion the following syntax is provided. The underlying types must be convertible according to C++ rules

```
STRONG_TYPE(radian_type,float,-3.14f,3.14f);
STRONG_TYPE(degree_type,float,-180f,180f);

radian_type radians = 2.0;
degree_type degrees = radian_type::cast_to<degree_type>(radians)/3.14*180;
```

Composite types

Arrays

- Strong types can be used to define array bounds

```
STRONG_ENUM(fish_type, cod, haddock, plaice, sole);
```

- As per C++ STL convention the stored type is defined first and the index type second
- The stored type can be a standard C++ type, or a strong::type, strong::array, strong::tuple etc.
- The index must be a strong::type, the underlying type can be an enumeration or integral but not a floating point type

```
STRONG_TYPE(velocity_type,float,0f,120000f);
using fish_speed_array_type = strong::array<velocity_type, fish_type>;
```

- The array instance can be accessed using the enumeration

```
fish_speed_array_type fish_speeds{2.3, 4.5, 1.7, 200.3};
velocity_type velocity = fish_speeds[fish_type_enum::haddock]; //4.5
```

Reference

Multidimensional Arrays

- The multiple ranges can be assigned to a single array. Under the bonnet this creates nested arrays of arrays

```
STRONG_ENUM(ocean_type, atlantic, pacific, indian, arctic);

using ocean_fs_array_type = strong::array<velocity_type,ocean_type,fish_type>;

fish_speed_array_type atlantic_speeds{2.3,6.5,1.7,200.3};
fish_speed_array_type pacific_speeds {4.4,4.5,1.5,20.3};
fish_speed_array_type indian_speeds  {6.1,2.5,1.2,2.3};
fish_speed_array_type artic_speeds   {1.3,0.5,7.8,0.3};

ocean_fs_array_type
ofs{atlantic_speeds,pacific_speeds,indian_speeds,artic_speeds};
velocity_type velocity = ofs[ocean_type_enum::indian][fish_type_enum::haddock];
//2.5
```

Unconstrained Arrays

- The definition of the index type may be deferred

```
template <typename index>
using velocity_array_type = strong::array<velocity_Type,index>;
```

- However the index still needs to be defined at compile time

Runtime Arrays

- The Strong runtime_array wraps std::vector
- std::vector uses dynamically allocated memory on the heap
- The wrapper limits the functionality so that heap space is only allocated on construction. The memory is freed at the end of the variable's lifetime.

```
using runtime_array_type = strong::runtime_array<velocity_type, fish_type>;
void example_func(fish_type first, fish_type last)
{
    strong::range<fish_type> array_range(first, last);
    runtime_array_type my_array(array_range);
}
```

Variants

- The strong variant wraps the std::variant to allow indexing by enum;
- To simplify the definition of a strong variant a macro is provided

```
STRONG_VARIANT(Var_type,4,
                     START,0,STOP,0,BIT,1,DEMAND,2,
                     uint8_t,       bool, double};

Var_type a,c;
Var_type b(Var_type_kind_enum::BIT, true);
c.set<Var_type_kind_enum::DEMAND>(3.1);
a=b;
std::cout << a.get<Var_type_kind_enum::BIT>() << std::endl;   //TRUE
a=c;
std::cout << a.get<Var_type_kind_enum::DEMAND>() << std::endl; //3.1
```

- The macro parameters are as follows;
  - Type name
  - Number of variants
  - Pairs of variant name and associated type position
  - List of associated types.
- The macro creates a strong enumerated type <Type name>_kind that uses an enum class called <Type name>_kind_enum.

Representation

Enums revisited

```
STRONG_ENUM(fish_type, cod, haddock, plaice, sole);
```

- To use Strong types with representation clauses there are also a macro variants to define the number of bits

```
STRONG_ENUM_SIZE(fish_type, 2, cod, haddock, plaice, sole);
```

- The above uses the default range of 0 to 3 to represent the enums. So they fit in the 2 bits defined. This is checked at compile time.

- It may be we need the enums to be represented by different values

```
STRONG_ENUM_REP(fish_type, 8, cod,      0b01010101,
                              haddock,  0b01010111,
                              plaice,   0b01011101,
                              sole,     0b01110101);
```

- In the macro the underlying enum class is left with the default range. The representation values live along side for serialisation and deserialization functions.

Tuple

- The strong variant wraps the std:tuple to allow indexing by enum;
- It also allows the specification of byte offsets and start/stop bits along with endianness

```
STRONG_TYPE_SIZE(nibble_type, 4, uint8_t,0,15)
STRONG_TYPE_SIZE(byte_type,   8, uint8_t,0,255)
STRONG_TUPLE (message_type, LITTLE,
                      header,  nibble_type, 0, 0,  3,
                      body,     byte_type,   0, 4, 11,
                      footer,  nibble_type, 1, 4,  7);

message_type message(0xE, 0xFD, 0xA);
uint8_t raw_msg[stong::numbytes(message_type::size())];
strong:: raw_c_ptr_container<uint8_t> raw_msg_wrp (raw_msg, 2);
Message.get_byte_array(raw_msg_wrp);
std::cout << std::hex << raw_msg_wrp[0] << std ::endl;              // FE
std::cout << std::hex << raw_msg_wrp[1] << std ::endl;              // AD
```

- Serialization (get_byte_array), de-serialization (set_from_byte_array) and validation functions are provided
- Functions to output the message description to a text file are also provided

Attributes

Type attributes

- The following constexpr functions are available for strong types
  - first, last
  - range
  - length
  - pos, val (discrete types only, Use with floating point types will raise an error at compile time.)
  - enum_rep, enum_val (as above but uses the representation values)
  - prev, succ (discrete types only, Use with floating point types will raise an error at compile time.)
  - size (in bits when used with tuple serialisation)

Lifetime checks

Access types

- The Strong Access provides a wrapper for the standard C pointer

```
using fish_acc_t = strong::access<fish_type>;

fish_type fish = fish_type_enum::plaice;
fish_type_acc_t fish_acc = &fish;
*fish_acc = fish_type_enum::haddock;
```

- Fish_acc can be used in the same way as a standard pointer. However, if it is used whilst null the last chance handler will be called.
- The Strong Reference provides a wrapper for the standard C++ reference

```
using fish_ref_t = strong::reference<fish_type>;

fish_type fish = fish_type_enum::plaice;
fish_type_ref_t fish_ref = fish;
*fish_ref = fish_type_enum::haddock;
```

- Note: unlike standard C++ references the strong reference requires de-referencing before use.

Lifetime checks

- The class strong::lifetime_check enables the lifetime track functionality.
- Any class that inherits this class will have its lifetime checked.
- Strong::lifetime_check can be inherited by user defined classes. It is inherited by the following 'strong' composite types.
  - strong::runtime_array
  - strong::tuple
  - strong::variant
- The Feature involves checks conducted in the destructor of types that inherit from strong::lifetime_check. This prevents objects of these types from being considered literal types and among other things declared constexpr. Therefore it is not applied to the basic "strong" library types.
- If on destruction pointers or references to the object remain the last chance handler is called.
- The lifetime check feature currently uses dynamic memory allocation to track the existence of strong::access & strong::reference types and where they point. This feature can be disabled using the compiler switch –DSTRONG_LIFETIME_CHECKS_OFF. It is also disabled as part of switching all checks off using –DSTRONG_CHECKS_OFF.

Pointer type conversion

```
class def_vehicle : public strong::lifetime_check{
};

class def_cars : public def_vehicle{
public:
   STRONG_ENUM(car_makes_type,ford,jaguar);
};


{
   strong::access<def_cars> my_car_ptr
   {
      def_cars my_car;
      my_car_ptr = &my_car;
      strong::access<def_vehicle> any_vehicle_ptr = my_car_ptr.convert<def_vehicle>();
   } // my_car goes out of scope but my_car_ptr is still set Last Chance Handler called.
}
```

Tracked Dynamic Memory

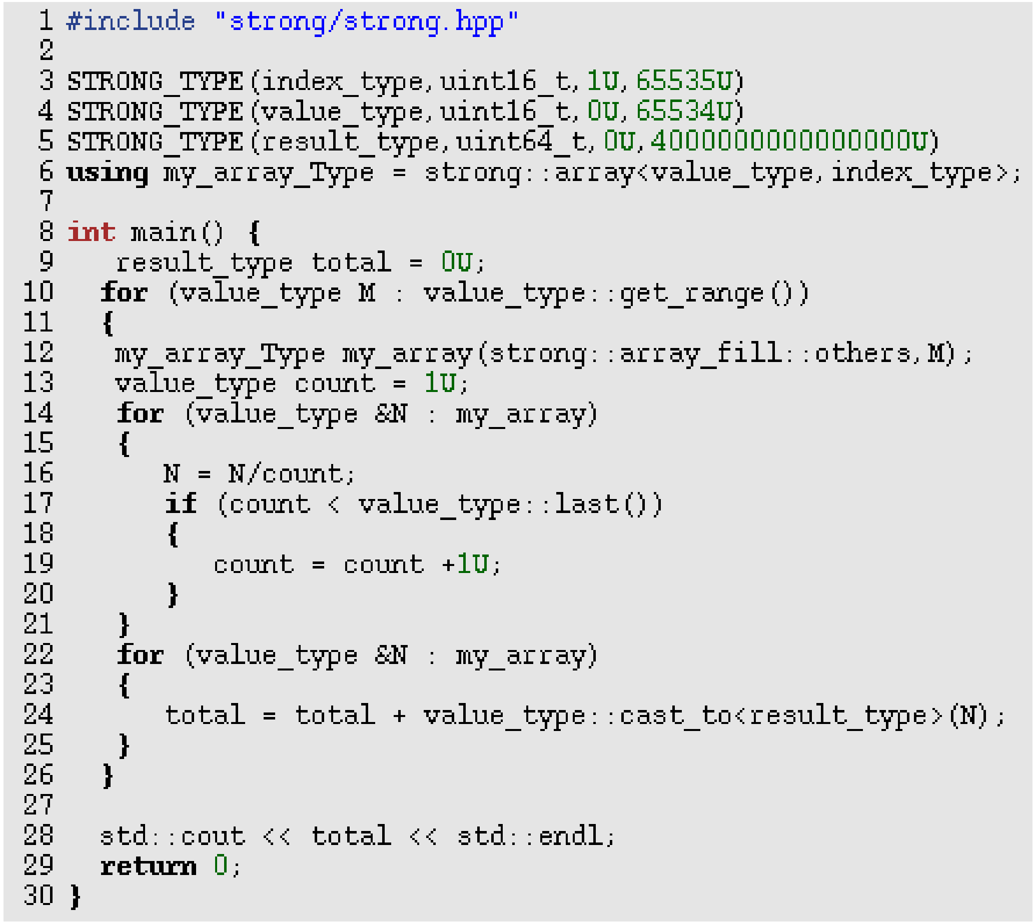
- The strong::access type provides memory allocation and deallocation routines. Dynamic memory allocation and deallocation raises the risk of various errors.
  - **Memory Leak:** If strong::access is instantiated using a class that inherits from strong::lifetime_check the allocated memory is tracked. If the last pointer to allocated memory goes out of scope or it reallocated before the object is deallocated then the last chance handler is called.
    - Note, if a dynamically allocated circular linked list were to become orphaned this would not be detected and a memory leak will occur.
  - **Repeat Deallocation:** If an object is inadvertently deallocated twice undefined behaviour would occur. This is detected before it happens and the last chance handler is called.
  - **Deallocation of inappropriate memory:** If an object that is not dynamically allocated is inadvertently deallocated undefined behaviour would occur. This is detected before it happens and the last chance handler is called.
  - **Dangling pointers:** The same protection is provided for dynamic memory as described for static memory above.
- The above protections are only maintained if the strong::access functions are used and native C++ pointer functions are avoided. This can only be checked by review.

```
const radian_type zero_rad = 0.0;
radian_acc_t rad_ptr1 = strong::access<radian_type>::allocate(3.1); // new radian_type on heap
//rad_ptr1 = &zero_rad;  // memory leak as nothing now points to object on heap
rad_ptr1.deallocate();   // memory freed and pointer set to nullptr.
//rad_ptr1.deallocate();   //trying to deallocate nullptr.
strong::access<radian_type> rad_ptr2 = &zero_rad;
rad_ptr2.deallocate();   //trying to deallocate stack
```

Compile time check example

Compile time check example

Example code

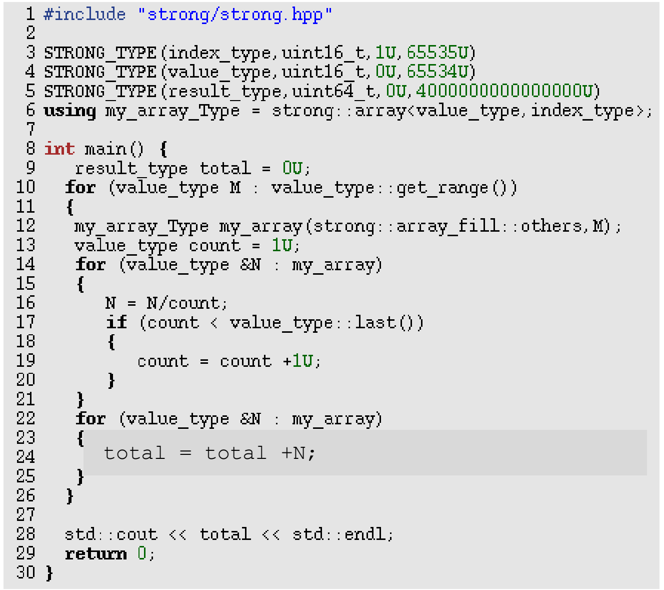


```cpp
#include "strong/strong.hpp"

STRONG_TYPE(index_type,uint16_t,1U,65535U)
STRONG_TYPE(value_type,uint16_t,0U,65534U)
STRONG_TYPE(result_type,uint64_t,0U,4000000000000000U)
using my_array_Type = strong::array<value_type,index_type>;

int main() {
   result_type total = 0U;
  for (value_type M : value_type::get_range())
  {
   my_array_Type my_array(strong::array_fill::others,M);
   value_type count = 1U;
   for (value_type &N : my_array)
   {
      N = N/count;
      if (count < value_type::last())
      {
         count = count +1U;
      }
   }
   for (value_type &N : my_array)
   {
      total = total + value_type::cast_to<result_type>(N);
   }
  }

  std::cout << total << std::endl;
  return 0;
}
```

Reference

Compile time check example

Compile time error



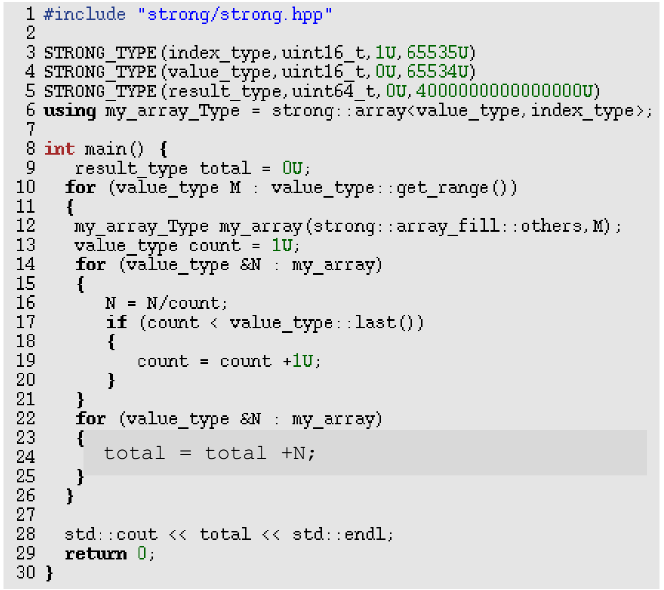
```
 1 #include "strong/strong.hpp"
 2
 3 STRONG_TYPE(index_type,uint16_t,1U,65535U)
 4 STRONG_TYPE(value_type,uint16_t,0U,65534U)
 5 STRONG_TYPE(result_type,uint64_t,0U,4000000000000000U)
 6 using my_array_Type = strong::array<value_type,index_type>;
 7
 8 int main() {
 9    result_type total = 0U;
10   for (value_type M : value_type::get_range())
11   {
12    my_array_Type my_array(strong::array_fill::others,M);
13    value_type count = 1U;
14    for (value_type &N : my_array)
15    {
16       N = N/count;
17       if (count < value_type::last())
18       {
19          count = count +1U;
20       }
21    }
22    for (value_type &N : my_array)
23    {
24      total = total +N;
25    }
26   }
27
28   std::cout << total << std::endl;
29   return 0;
30 }
```

```
/include/strong/type.hpp: In instantiation of 'static constexpr void strong::hidden_type<T, P_range, rep_clause, to_string_ptr>::is_compatible_type()[with
other_type = strong::hidden_type<short usigned int, value_type_def::value_type_range,0,0>; T = long long insigned int; const strong range<T>& P_Range
= result_type_def::result_type_range; const strong::hidden_enum_rep<T, P_range>* rep_clause = 0; const string (* to_string_ptr)(T) = 0]':
/include/strong/type.hpp:843:68: required from 'constexpr strong::hidden_type<T, P_range, rep_clause, to_string_ptr> strong::hidden_type<T, P_range,
rep_clause, to_string_ptr>::operator+(other_type&&) const [with other_type = strong::hidden_type<short usigned int,
value_type_def::value_type_range,0,0>&; T= long long insigned int; const strong range<T>& P_Range = result_type_def::result_type_range; const
strong::hidden_enum_rep<T, P_range>* rep_clause = 0; const string (* to_string_ptr)(T) = 0]'
/timing_test/Execution_time.cpp:24:23: required from here
/include/strong/type.hpp:487:44 error: static assertion failed: is compatible : incompatible base type
487 |          static_assert(std::is_same_v<typename hidden_type::local_range::type,
    |                              ~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
488 |                                        typename other_type::local_range::type>,
```

Reference

Compile time check example

Compile time error



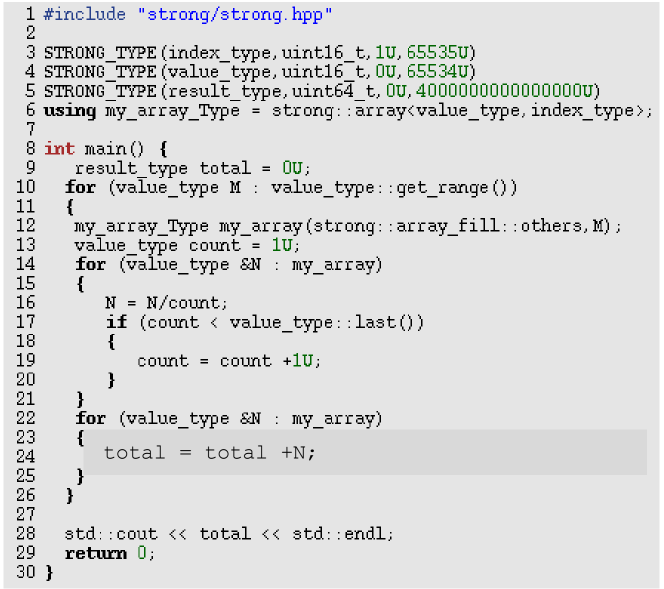
```
 1 #include "strong/strong.hpp"
 2
 3 STRONG_TYPE(index_type,uint16_t,1U,65535U)
 4 STRONG_TYPE(value_type,uint16_t,0U,65534U)
 5 STRONG_TYPE(result_type,uint64_t,0U,4000000000000000U)
 6 using my_array_Type = strong::array<value_type,index_type>;
 7
 8 int main() {
 9    result_type total = 0U;
10   for (value_type M : value_type::get_range())
11   {
12    my_array_Type my_array(strong::array_fill::others,M);
13    value_type count = 1U;
14    for (value_type &N : my_array)
15    {
16       N = N/count;
17       if (count < value_type::last())
18       {
19          count = count +1U;
20       }
21    }
22    for (value_type &N : my_array)
23    {
24       total = total +N;
25    }
26   }
27
28   std::cout << total << std::endl;
29   return 0;
30 }
```

```
/include/strong/type.hpp: In instantiation of 'static constexpr void strong::hidden_type<T, P_range, rep_clause, to_string_ptr>::is_compatible_type()[with
other_type = strong::hidden_type<short usigned int, value_type_def::value_type_range,0,0>; T = long long insigned int; const strong range<T>& P_Range
= result_type_def::result_type_range; const strong::hidden_enum_rep<T, P_range>* rep_clause = 0; const string (* to_string_ptr)(T) = 0]':
/include/strong/type.hpp:843:68: required from 'constexpr strong::hidden_type<T, P_range, rep_clause, to_string_ptr> strong::hidden_type<T, P_range,
rep_clause, to_string_ptr>::operator+(other_type&&) const [with other_type = strong::hidden_type<short usigned int,
value_type_def::value_type_range,0,0>&; T= long long insigned int; const strong range<T>& P_Range = result_type_def::result_type_range; const
strong::hidden_enum_rep<T, P_range>* rep_clause = 0; const string (* to_string_ptr)(T) = 0]'
/timing_test/Execution_time.cpp:24:23: required from here
/include/strong/type.hpp:487:44 error: static assertion failed: is compatible : incompatible base type
487 |          static_assert(std::is_same_v<typename hidden_type::local_range::type,
    |                                   ~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
488 |                                                      typename other_type::local_range::type>,
```

Reference

Compile time check example

Compile time error



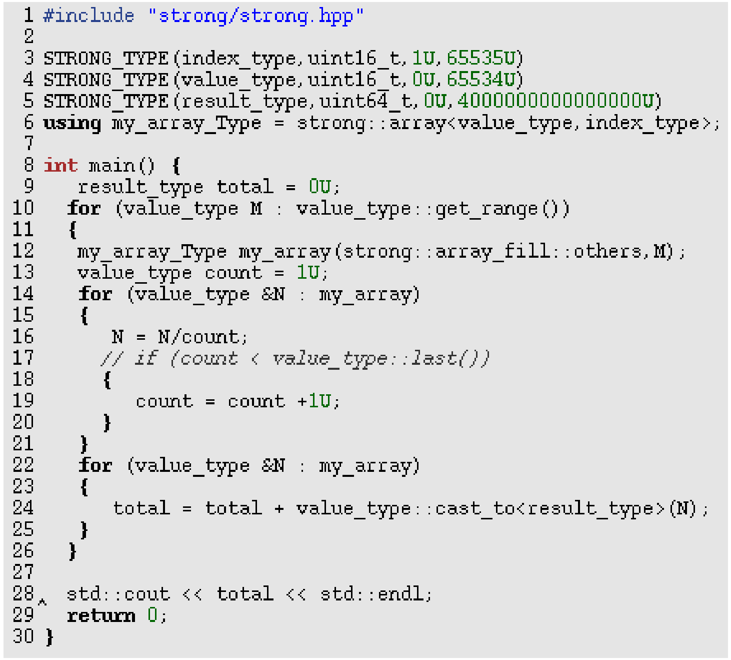
```
 1 #include "strong/strong.hpp"
 2
 3 STRONG_TYPE(index_type,uint16_t,1U,65535U)
 4 STRONG_TYPE(value_type,uint16_t,0U,65534U)
 5 STRONG_TYPE(result_type,uint64_t,0U,4000000000000000U)
 6 using my_array_Type = strong::array<value_type,index_type>;
 7
 8 int main() {
 9     result_type total = 0U;
10    for (value_type M : value_type::get_range())
11    {
12     my_array_Type my_array(strong::array_fill::others,M);
13     value_type count = 1U;
14     for (value_type &N : my_array)
15     {
16        N = N/count;
17        if (count < value_type::last())
18        {
19           count = count +1U;
20        }
21     }
22     for (value_type &N : my_array)
23     {
24       total = total +N;
25     }
26    }
27
28    std::cout << total << std::endl;
29    return 0;
30 }
```

```
/include/strong/type.hpp: In instantiation of 'static constexpr void strong::hidden_type<T, P_range, rep_clause, to_string_ptr>::is_compatible_type()[with other_type = strong::hidden_type<short usigned int, value_type_def::value_type_range,0,0>; T = long long insigned int; const strong range<T>& P_Range = result_type_def::result_type_range; const strong::hidden_enum_rep<T, P_range>* rep_clause = 0; const string (* to_string_ptr)(T) = 0]':
/include/strong/type.hpp:843:68: required from 'constexpr strong::hidden_type<T, P_range, rep_clause, to_string_ptr> strong::hidden_type<T, P_range, rep_clause, to_string_ptr>::operator+(other_type&&) const [with other_type = strong::hidden_type<short usigned int, value_type_def::value_type_range,0,0>&; T= long long insigned int; const strong range<T>& P_Range = result_type_def::result_type_range; const strong::hidden_enum_rep<T, P_range>* rep_clause = 0; const string (* to_string_ptr)(T) = 0]'
/timing_test/Execution_time.cpp:24:23: required from here
/include/strong/type.hpp:487:44 error: static assertion failed: is compatible : incompatible base type
487 |          static_assert(std::is_same_v<typename hidden_type::local_range::type,
    |                                 ~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
488 |                                                   typename other_type::local_range::type>,
```

Reference

Runtime check example

Runtime error (no optimisation)



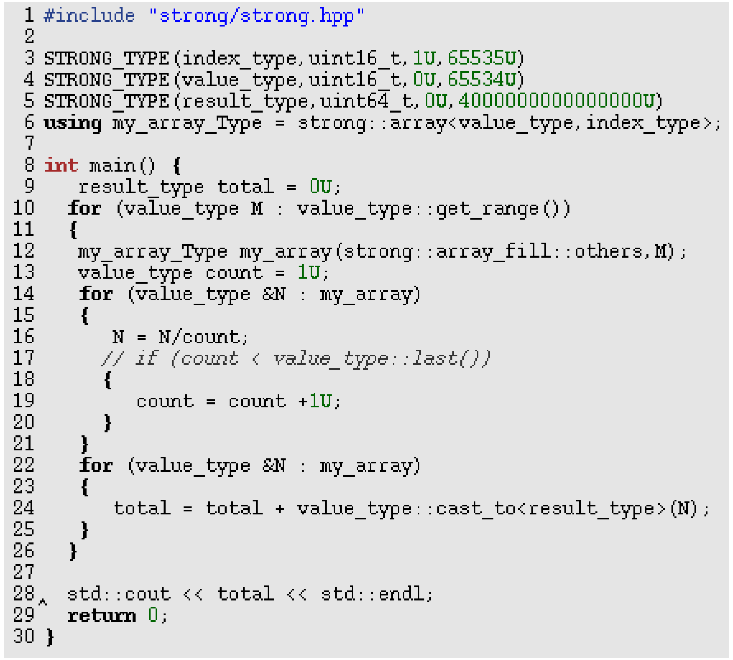
```
 1 #include "strong/strong.hpp"
 2
 3 STRONG_TYPE(index_type,uint16_t,1U,65535U)
 4 STRONG_TYPE(value_type,uint16_t,0U,65534U)
 5 STRONG_TYPE(result_type,uint64_t,0U,4000000000000000U)
 6 using my_array_Type = strong::array<value_type,index_type>;
 7
 8 int main() {
 9    result_type total = 0U;
10   for (value_type M : value_type::get_range())
11   {
12    my_array_Type my_array(strong::array_fill::others,M);
13    value_type count = 1U;
14    for (value_type &N : my_array)
15    {
16       N = N/count;
17      // if (count < value_type::last())
18      {
19         count = count +1U;
20      }
21    }
22    for (value_type &N : my_array)
23    {
24       total = total + value_type::cast_to<result_type>(N);
25    }
26   }
27
28   std::cout << total << std::endl;
29   return 0;
30 }
```

```
"addition value out of range (1)"
 0# 0x0804D27A in ./obj_on/Execution_time
 1# 0x0804FBC6 in ./obj_on/Execution_time
 2# 0x0804F352 in ./obj_on/Execution_time
 3# 0x0804EAD4 in ./obj_on/Execution_time
 4# 0x0804E4C6 in ./obj_on/Execution_time
 5# 0x0804D1B0 in ./obj_on/Execution_time
 6# 0x0804B11E in ./obj_on/Execution_time
 7# 0x0804BE21 in ./obj_on/Execution_time
 8# 0x0804C5E8 in ./obj_on/Execution_time
 9# 0x0804BEDA in ./obj_on/Execution_time
10# 0x0804AD89 in ./obj_on/Execution_time
11# __libc_start_main in /lib/libc.so.6
12# 0x0804AB61 in ./obj_on/Execution_time
```

```
/strong/src/error_handler.cpp:81
/apps/gnat_pro/22.2/x86_32/include/c++/10.3.1/bits/invoke.h:60
/apps/gnat_pro/22.2/x86_32/include/c++/10.3.1/bits/invoke.h:116
/apps/gnat_pro/22.2/x86_32/include/c++/10.3.1/bits/std_function.h:292
/apps/gnat_pro/22.2/x86_32/include/c++/10.3.1/bits/std_function.h:622
/strong/src/error_handler.cpp:58
/strong/include/strong/error_handler.hpp:89 (discriminator 6)
/strong/include/strong/type.hpp:858
/strong/include/strong/error_handler.hpp:73
/strong/include/strong/type.hpp:858 (discriminator 1)
/examples/timing_test/Execution_time.cpp:19 (discriminator 3)
??:?
```

Runtime check example

Runtime error (-O2)



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

STRONG_TYPE(index_type,uint16_t,1U,65535U)
STRONG_TYPE(value_type,uint16_t,0U,65534U)
STRONG_TYPE(result_type,uint64_t,0U,4000000000000000U)
using my_array_Type = strong::array<value_type,index_type>;

int main() {
   result_type total = 0U;
  for (value_type M : value_type::get_range())
  {
   my_array_Type my_array(strong::array_fill::others,M);
   value_type count = 1U;
   for (value_type &N : my_array)
   {
      N = N/count;
     // if (count < value_type::last())
     {
        count = count +1U;
     }
   }
   for (value_type &N : my_array)
   {
      total = total + value_type::cast_to<result_type>(N);
   }
  }

  std::cout << total << std::endl;
  return 0;
}
```

```
"addition value out of range (1)"
 0# 0x0804A2EB in ./obj_on/Execution_time
 1# 0x0804A3FD in ./obj_on/Execution_time
 2# 0x080495C6 in ./obj_on/Execution_time
 3# __libc_start_main in /lib/libc.so.6
 4# 0x08049AE1 in ./obj_on/Execution_time
```

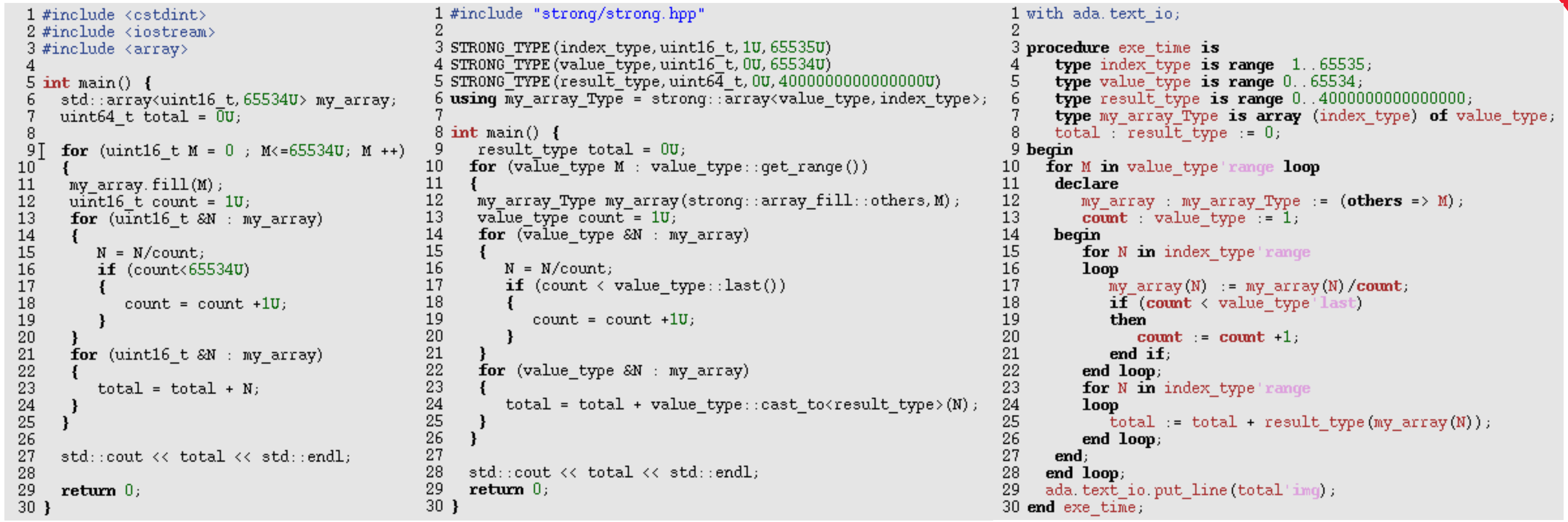
/apps/gnat_pro/22.2/x86_32/include/c++/10.3.1/bits/stl_vector.h:919
/strong/src/error_handler.cpp:72
/apps/gnat_pro/22.2/x86_32/include/c++/10.3.1/bits/basic_string.h:195
??:?

- Some work to do here

Reference

Performance

Execution time comparison



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

int main() {
  std::array<uint16_t,65534U> my_array;
  uint64_t total = 0U;

  for (uint16_t M = 0 ; M<=65534U; M ++)
   {
    my_array.fill(M);
    uint16_t count = 1U;
    for (uint16_t &N : my_array)
     {
       N = N/count;
       if (count<65534U)
       {
          count = count +1U;
       }
     }
    for (uint16_t &N : my_array)
     {
       total = total + N;
     }
   }

  std::cout << total << std::endl;

  return 0;
}
```

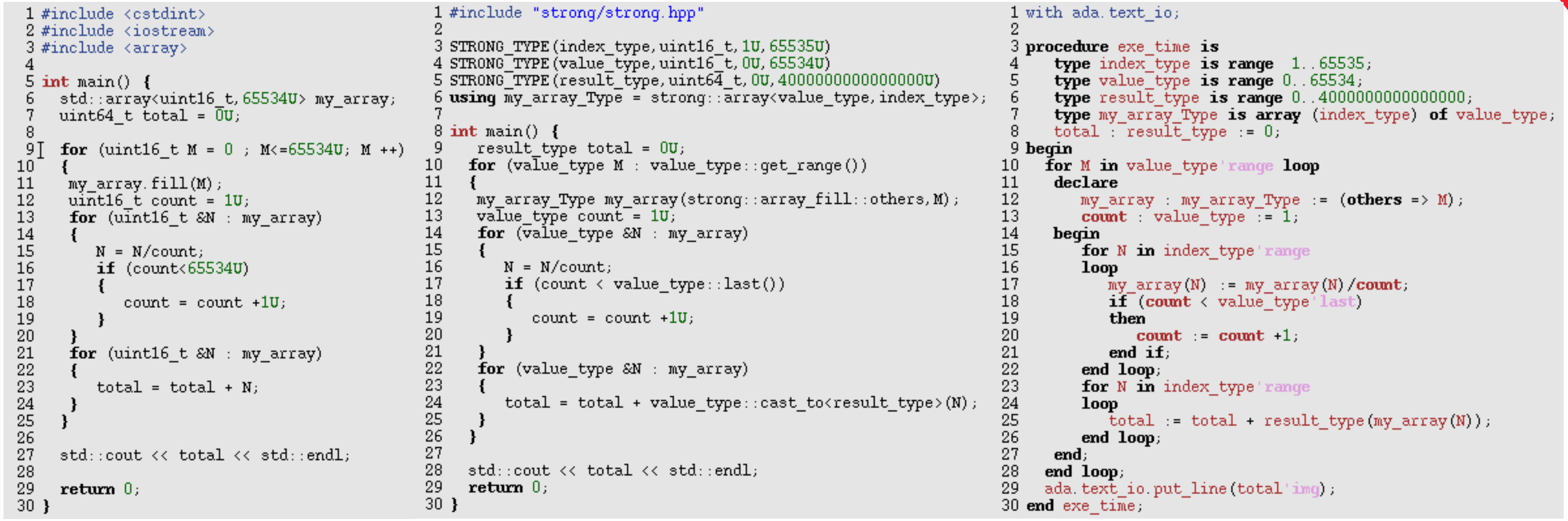
```
#include "strong/strong.hpp"

STRONG_TYPE(index_type,uint16_t,1U,65535U)
STRONG_TYPE(value_type,uint16_t,0U,65534U)
STRONG_TYPE(result_type,uint64_t,0U,4000000000000000U)
using my_array_Type = strong::array<value_type,index_type>;

int main() {
   result_type total = 0U;
  for (value_type M : value_type::get_range())
  {
   my_array_Type my_array(strong::array_fill::others,M);
   value_type count = 1U;
   for (value_type &N : my_array)
   {
      N = N/count;
      if (count < value_type::last())
      {
         count = count +1U;
      }
   }
   for (value_type &N : my_array)
   {
      total = total + value_type::cast_to<result_type>(N);
   }
  }

  std::cout << total << std::endl;
  return 0;
}
```

```
with ada.text_io;

procedure exe_time is
   type index_type is range  1..65535;
   type value_type is range 0..65534;
   type result_type is range 0..4000000000000000;
   type my_array_Type is array (index_type) of value_type;
   total : result_type := 0;
begin
  for M in value_type'range loop
   declare
      my_array : my_array_Type := (others => M);
      count : value_type := 1;
   begin
      for N in index_type'range
      loop
         my_array(N) := my_array(N)/count;
         if (count < value_type'last)
         then
            count := count +1;
         end if;
      end loop;
      for N in index_type'range
      loop
         total := total + result_type(my_array(N));
      end loop;
   end;
  end loop;
  ada.text_io.put_line(total'img);
end exe_time;
```

	Vanilla C++	Strong C++ Checks On	Strong C++ Checks Off	Ada Checks on	Ada Checks off
No Optimisation	46.06	1027.94	911.48	36.55	34.54
-O2	19.01	37.07	22.01	26.10	21.59

Reference

Compile time comparison



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

int main() {
  std::array<uint16_t,65534U> my_array;
  uint64_t total = 0U;

  for (uint16_t M = 0 ; M<=65534U; M ++)
  {
   my_array.fill(M);
   uint16_t count = 1U;
   for (uint16_t &N : my_array)
   {
      N = N/count;
      if (count<65534U)
      {
         count = count +1U;
      }
   }
   for (uint16_t &N : my_array)
   {
      total = total + N;
   }
  }

  std::cout << total << std::endl;

  return 0;
}
```

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

STRONG_TYPE(index_type,uint16_t,1U,65535U)
STRONG_TYPE(value_type,uint16_t,0U,65534U)
STRONG_TYPE(result_type,uint64_t,0U,4000000000000000U)
using my_array_Type = strong::array<value_type,index_type>;

int main() {
   result_type total = 0U;
  for (value_type M : value_type::get_range())
  {
   my_array_Type my_array(strong::array_fill::others,M);
   value_type count = 1U;
   for (value_type &N : my_array)
   {
      N = N/count;
      if (count < value_type::last())
      {
         count = count +1U;
      }
   }
   for (value_type &N : my_array)
   {
      total = total + value_type::cast_to<result_type>(N);
   }
  }

  std::cout << total << std::endl;
  return 0;
}
```

```ada
with ada.text_io;

procedure exe_time is
   type index_type is range  1..65535;
   type value_type is range 0..65534;
   type result_type is range 0..4000000000000000;
   type my_array_Type is array (index_type) of value_type;
   total : result_type := 0;
begin
  for M in value_type'range loop
   declare
      my_array : my_array_Type := (others => M);
      count : value_type := 1;
   begin
      for N in index_type'range
      loop
         my_array(N) := my_array(N)/count;
         if (count < value_type'last)
         then
            count := count +1;
         end if;
      end loop;
      for N in index_type'range
      loop
         total := total + result_type(my_array(N));
      end loop;
   end;
  end loop;
  ada.text_io.put_line(total'img);
end exe_time;
```

	Vanilla C++	Strong C++ Checks On	Strong C++ Checks Off	Ada Checks on	Ada Checks off
No Optimisation	1.61	9.03	8.42	2.04	1.75
-O2	1.85	11.71	8.56	1.55	2.07

Reference

Call for contribution

Open source?

- Aim to make the library freely available
- We are in the Defence industry, generally, we don’t release anything
- Need help in generating the governance model etc.

A stepping stone?

- Can we use this library to expose the wider C++ community to the benefits of these features?
- Could this encourage the take up of a language dedicated to their robust efficient implementation.