



Strongly Typed Declarative Bitsets in C++17

improving type safety

Ewoud Van Craeynest
Immanuel Litzroth

Strongly Typed Declarative Bitsets in C++17

motivation

```
1 foo(const bool check_authority,
2      const bool validate_user,
3      const bool launch_nukes);
4
5 foo(0, true, 42); // boom!
```

Strongly Typed Declarative Bitsets in C++17

motivation

the C++ standard states:

- bool is an integral type ([basic.types])
- bool values participate in integral promotions and conversions ([conv.bool])

```
1 foo(const bool check_authority,
2      const bool validate_user,
3      const bool launch_nukes);
4
5 foo(0, true, 42); // boom!
```

Strongly Typed Declarative Bitsets in C++17

motivation

Unrelated information represented by only a few integral types

- unrelated information
 - bool != character != integer
- regardless of the domain, quantity, unit, ...
 - puppies, distance, volume, time, force, ...
- lacking
 - semantics
 - clear (non-)relationships

Strongly Typed Declarative Bitsets in C++17

Unrelated information represented by only a few integral types

Posing some challenges:

- refactoring/understanding legacy code
- api boundaries
- representing which unit, if any
- conversions not checked the way we'd like
- integer promotion [conv.prom]
- integer conversion [conv.integral]
- boolean conversions [conv.bool]

Looks like we're in need of some **type safety**

Strongly Typed Declarative Bitsets in C++17

type safety

Not a novel need

- famous mishaps from the past
- Code Smells
- C++ Core Guidelines
- boost/std::chrono
- boost::units
 - BOOST_STRONG_TYPEDEF
 - boost::format
- foonathan/type_safe
- robertramey/safe_numerics
 - (std proposal & boost incubator)
- joboccara/NamedType
- nholthaus/units
- wardw/simpleunit

type safety

Code Smells

- Magic Numbers
- Primitive Obsession
- Long Parameter List
- Type Embedded in Name
- Shotgun Surgery

type safety

C++ Core Guidelines

- I.4: Make interfaces precisely and strongly typed
- I.24: Avoid adjacent unrelated parameters of the same type

```
1 foo(const bool check_authority,
2     const bool validate_user,
3     const bool launch_nukes);
4
5 foo(0, true, 42); // boom!
```

type safety

std::chrono

- compile time type safety
- chrono::literals

```
1 sleep(10ms);
2 seconds s{42};
3
4 seconds s = 42; // error: conversion from 'int' to non-scalar type 'std::chrono::seconds'
5 // {aka 'std::chrono::duration<long long int>'} requested
6
7 seconds s = 4200ms; // error: conversion from 'duration<[...],ratio<[...],1000>>'
8 //to non-scalar type 'duration<[...],ratio<[...],1>>' requested
9
10 seconds s = duration_cast<seconds>{4200ms}; // truncated to 4s
```

type safety

boost::units

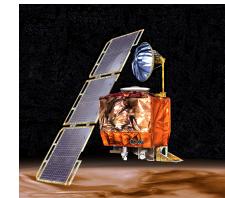
concepts like: dimension, unit, system, quantity

```
1 #include <boost/units/systems/si/energy.hpp>
2 #include <boost/units/systems/si/force.hpp>
3 #include <boost/units/systems/si/length.hpp>
4
5 using namespace boost::units;
6 using namespace boost::units::si;
7
8 quantity<force> f(2.0 * newton); // Define a quantity of force.
9 quantity<length> dx(2.0 * meter); // and a distance,
10 quantity<energy> e(work(f,dx)); // and calculate the work done.
```

The need for type safety

famous examples

- Patriot Missile Failure (1991)
 - multiplied 24-chopping
 - a case for std::chrono and its meta code?
- Ariane5 explosion (1996)
 - 64-bit float to 16-bit integer
 - loss of precision in conversion
- Mars Climate Orbiter disintegration (1999)
 - one team worked with pounds, another with Newtons
 - different units for a given physical quantity



Strongly Typed Declarative Bitsets in C++17

talk content

- numbers are just numbers, can't blame them for our mistakes
 - any information can be encoded in a number of bits
- multiple efforts going on in our ecosystem on safer numerics
 - allowing us to express **what** is encoded
- and today, it's about a smaller member of that family, the **bit**
 - they suffer from this too
 - and perhaps haven't gotten as much attention

Strongly Typed Declarative Bits in C++17

talk outline

- Typed boolean
- Bitfield offset
- Strongly typed declarative bitset
- Looking to the future



Typed boolean

Typed boolean

motivation

- we'd like to be able to express ourselves as follows
- without leaving room for error

```
1 foo(CheckAuthority,
2     ValidateUser,
3     LaunchNukes);
4
5 foo(0, true, 42);      // error!
6
7 foo(UseCache{true},   // error!
8     ValidateUser{false},
9     LaunchTheNukes{false});
```

Typed boolean

motivation

```
1 foo(CheckAuthority,  
2     ValidateUser,  
3     LaunchNukes);  
4  
5 foo(0, true, 42);      // error!  
6  
7 foo(UseCache{true},   // error!  
8     ValidateUser{false},  
9     LaunchTheNukes{false});
```

- code is easier to read
- no overly elaborate naming schemes needed
- more confident refactoring
- no compilation overhead

Typed boolean implementation

- our options
 - preprocessor based class generation
 - tag-based template class instantiations
 - enum class with base bool
 - others we've overlooked
- aim for zero-overhead abstraction
 - Modern C++ after all

Typed boolean implementation

- use enum class to generate distinct new types
- use enum-base type bool
- define them without enumerator-list
- initializing a enum class with boolean-like object (C++17)

```
1 enum class Aboolean : bool {};
2
3 Aboolean booll{true};
```

Typed boolean implementation

- avoid boilerplate code
- express intent

```
1 #define DECLARE_TYPED_BOOLEAN(name) enum class name : bool;  
2  
3 DECLARE_TYPED_BOOLEAN(Aboolean);
```

Typed boolean implementation

avoid more boilerplate code beyond the declaration

```
1 #ifndef TYPED_BOOLEAN_HPP
2 #define TYPED_BOOLEAN_HPP
3
4 #include <iostream>
5 #define DECLARE_TYPED_BOOLEAN(name) enum class name : bool;
6
7 #define DEFINE_TYPED_BOOLEAN(name)
8     enum class name : bool {};
9
10 inline std::ostream& operator<<(std::ostream& os, const name& in)
11 {
12     return os << (in == name{true} ? (#name "::true") : (#name "::false"));
13 }
14
15 #endif // TYPED_BOOLEAN_HPP
```

Typed boolean

usage

- generate a new type
- use the type like any other

```
1 DEFINE_TYPED_BOOLEAN(ABoolean);
2
3 int main()
4 {
5     std::cout << "size of bool is " << sizeof(bool) << std::endl;
6     std::cout << "size of ABoolean is: " << sizeof(ABoolean) << std::endl;
7
8     ABoolean bool1{false};
9     std::cout << "bool1 is: " << bool1 << std::endl;
10
11    ABoolean bool2{true};
12    std::cout << "bool2 is: " << bool2 << std::endl;
13 }
```

Typed boolean

usage & type safety

- accept it like any other type
- explicitly convert to its unsafe type where needed

```
1 void test_a_boolean(ABoolean a_boolean)
2 {
3     if(static_cast<bool>(a_boolean))
4     {
5         std::cout << "passed boolean is considered true: " << a_boolean << std::endl;
6     }
7     else
8     {
9         std::cout << "Passed boolean is considered false: " << a_boolean << std::endl;
10    }
11 }
```

Typed boolean

type safety

strong typing is enforced

```
1 // error: narrowing conversion of '20' from 'int' to 'bool' inside { } [-Wnarrowing]
2 ABoolean bool3{20};
3
4 // error: narrowing conversion of '-1' from 'int' to 'bool' inside { } [-Wnarrowing]
5 ABoolean bool3{-1};
6
7 // cannot convert 'bool' to 'ABoolean' for argument '1' to 'void test_a_boolean(ABoolean)'
8 test_a_boolean(true);
9
10 // cannot convert 'int' to 'ABoolean' for argument '1' to 'void test_a_boolean(ABoolean)'
11 test_a_boolean(1);
```

Typed boolean overhead

- everyone likes zero- or near-zero-overhead abstractions
- how does this one fare?



Typed boolean

Godbolt compiler explorer

gcc-8.1 -O2 --std=c++17

```
1 int f1(const bool arg)
2 {
3     if(arg)
4     {
5         return 10;
6     }
7     else
8     {
9         return 15;
10    }
11 }
```

```
1 f1(bool):
2     cmp dil, 1
3     sbb eax, eax
4     and eax, 5
5     add eax, 10
6     ret
```

Typed boolean

Godbolt compiler explorer

gcc-8.1 -O2 --std=c++17

```
1 int f2(const ATypedBoolean arg)
2 {
3     if(static_cast<bool>(arg))
4     {
5         return 10;
6     }
7     else
8     {
9         return 15;
10    }
11 }
```

```
1 f2(ATypedBoolean):
2     cmp dil, 1
3     sbb eax, eax
4     and eax, 5
5     add eax, 10
6     ret
```

```
1 f1(bool):
2     cmp dil, 1
3     sbb eax, eax
4     and eax, 5
5     add eax, 10
6     ret
```

Typed boolean

some disadvantages

- `static_cast` needed in boolean context
 - no way to make user defined conversion from enum class to another type
- unintended conversions
 - not 100% strong, it would seem
 - oversight in the standard

```
1 Validate v { true };
2 LaunchTheNukes ltn { v }; // not cool!
```

Fun fact

Ólafur's fruit salad

by Ólafur Waage

```
1 enum class Orange{};
2 enum class Apple{};
3
4 int main()
5 {
6     Orange o{4};
7     Apple a{3};
8     Apple x{o}; // Oops
9     // Apple y = o; // Fails
10
11    return 0;
12 }
```

Typed boolean

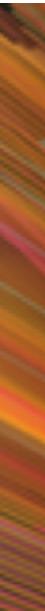
summary

- enum class with enum-base bool & no values
- zero-overhead abstraction
- static_cast needed



Bitset

©2018 Western Digital Corporation or its affiliates. All rights reserved.



Bitset

motivation

- want to keep bits around
- want to store them efficiently

Bitset

motivation

struct-of-bool space overhead

- waste $(\text{sizeof}(\text{bool}) * \text{CHAR_BIT}) - 1$ bits per bool
- bad Spacial Locality of Reference

```
1 DEFINE_TYPED_BOOLEAN(UseCache);
2 DEFINE_TYPED_BOOLEAN(ValidateUser);
3 DEFINE_TYPED_BOOLEAN(LaunchTheNukes);
4
5 struct Doom {
6     UseCache cache;
7     ValidateUser validate;
8     LaunchTheNukes launch;
9 };
10
11 sizeof(Foo);
```

Bitset

enter std::bitset

- fixed-size sequence of N bits
- efficiently packed, but **type unsafe**
 - more precisely: unsafe position into the set
 - did you mean to manipulate that bit?
 - or the one next to it?
 - or one from another bitset?
 - did I even pass the right bitset?

```
1 std::bitset<3> doom;      // fits in any single word
2 doom.flip(1);            // or did I mean 0?
3
4 doom.test(day_of_week); // whoops, wrong domain
```

Bitset

typical implementation

```
1 enum class Flags : uint8_t
2 {
3     fieldOne = 1,
4     fieldTwo = 1 << 1,
5     fieldThree = 1 << 2,
6     // skip any unused bits in this word
7     fieldFour = 1 << 5,
8 };
9
10 somebitset.set(DifferentFlags.meh); // no type safety
```

Bitset

versus typed boolean

- we want the type safety offered by typed boolean
- with the space efficiency of `std::bitset`

```
1 std::bitset<3> doom;
2 doom.flip(1);           // how to get an error for these situations?
3 doom.test(day_of_week);
```

Bitset

choosing is loosing

- we want the type safety offered by typed boolean
- **with** the space efficiency of std::bitset

```
1 doom.test<doom::LaunchTheNukes>();
2 doom.set(doom::LaunchTheNukes{false},
3           doom::CheckAuthority{true});
4
5 doom.test<42>(); // error: no matching function for call to 'Doom::test<42>()'
```

Bitset

we want ...

```
1 MAKE_BITFIELD(Doom, CheckAuthority, ValidateUser, LaunchTheNukes);
2 MAKE_BITFIELD(Foo, SkipValidation, SkipMonday, SkipMeeting);
3
4 Doom doom;
5 Foo foo;
6
7 doom.set(Foo::SkipMeeting{false}); // error: 'val' is not a member of
8 // 'Doom::BitFieldOffset<Foo_detail::SkipMeeting>'
9 // {aka 'Doom_detail::BitFieldOffset<Foo_detail::SkipMeeting>'}
10
11 doom.test<Foo::SkipValidation>(); // error: 'val' is not a member of
12 // 'Doom::BitFieldOffset<Foo_detail::SkipValidation>'
13 // {aka 'Doom_detail::BitFieldOffset<Foo_detail::SkipValidation>'}
```

Bitset

we want ...

```
1 class Doom : private std::bitset<2>
2 {
3     // ...
4 public:
5     using CheckAuthority = Doom_detail:: CheckAuthority;
6     using ValidateUser = Doom_detail:: ValidateUser;
7     using LaunchTheNukes = Doom_detail:: LaunchTheNukes;
8
9     template<typename...Args>
10    TestBitfield(Args... args){ set(args...); }
11
12    template<typename BitFieldBool> constexpr
13    BitFieldBool test() const {
14        return BitFieldBool(
15            this_bitset_type::operator[]( BitFieldOffset<BitFieldBool>::val) );
16    }
17
18    template<typename... BitFieldBools>
19    void set(BitFieldBools... ) {
20        (... , set_bitfield(args));
21    }
22};
```

Bitset

we want safety

- need a way to make the position into a bitset **type safe**
- need a way to tie the type of a bit to an ordinary position number

```
1 template<typename BitFieldBool> constexpr
2 BitFieldBool test() const {
3     return BitFieldBool(
4         this_bitset_type::operator[]( BitFieldOffset<BitFieldBool>::val ) );
5 }
```

Bitfield offset

bitset with strong position types

- need a way to make the position into a bitset **type safe**
- need a way to tie the type of a bit to an ordinary position number

```
1 #define MAKE_INDEX_FOR_BOOLEAN(r, data, i, newtype)
2     template<>
3     struct BitFieldOffset<newtype>
4     {
5         static constexpr size_t val = i;
6     };

```

Strongly Typed Declarative Bitsets in C++17

putting it all together

- we got some new building blocks
 - typed boolean
 - bitfield offset
 - some boiler plate code
- would be nice to have something of a distinct-bitset-type-generator
- to define the required strong bitset types easily

```
1 MAKE_BITFIELD(Doom, CheckAuthority, ValidateUser, LaunchTheNukes);
```

Strongly Typed Declarative Bits in C++17

putting it all together

we require:

- BitFieldOffset specialisation
- typed boolean type generation
- reuse of std::bitset
- exposure of our strong boolean types
- type safe accessors and mutators

Strongly Typed Declarative Bitsets in C++17

hiding implementational details

- need to define strong bool and offset type somewhere privately, sort of
- use a `details` namespace, unique to a bitset

```
1 #define MAKE_BITFIELDS(name, ...)  
2     namespace BOOST_PP_CAT(name, _detail)  
3     {  
4         BOOST_PP_SEQ_FOR_EACH(MAKE_BOOLEAN_FOR_BITFIELD,  
5             BOOST_PP_EMPTY(),  
6             BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__));  
7  
8         template<typename T>  
9         struct BitFieldOffset  
10        {};  
11         BOOST_PP_SEQ_FOR_EACH_I(MAKE_INDEX_FOR_BOOLEAN,  
12             BOOST_PP_EMPTY,  
13             BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__));  
14     }
```

Strongly Typed Declarative Bitsets in C++17

making a new bitset type

- derive from std::bitset
- expose strong bool types belonging to the set

```
1 // continuation of MAKE_BITFIELDS macro
2 class name :
3     private std::bitset<BOOST_PP_SEQ_SIZE(BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__))>
4     {
5     public:
6         BOOST_PP_SEQ_FOR_EACH(MAKE_BITFIELD_USING_DECLARATION,
7                               name,
8                               BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__));
9
10    template<typename...Args>
11    name(Args... args) {
12        set(args...);
13    }
14    // ...
```

Strongly Typed Declarative Bitsets in C++17

implementing members using modern techniques

- type safe, order agnostic set function
- using C++17 **fold expression**
 - earlier version used recursion and parameter pack peeling (C++11)

```
1 // continuation of MAKE_BITFIELDS macro
2 private:
3     template<typename BitFieldBool>
4     void bitfield_set(BitFieldBool in) {
5         this_bitset_type::set(
6             BitFieldOffset<BitFieldBool>::val, static_cast<bool>(in));
7     }
8
9 public:
10    template<typename... Args>
11    void set(Args... args) {
12        (..., bitfield_set(args));
13    }
```

Strongly Typed Declarative Bits in C++17

implementing members using modern techniques

type safe, test function

```
1 // continuation of MAKE_BITFIELDS macro
2 template<typename BitFieldBool> constexpr
3 BitFieldBool test() const {
4     return BitFieldBool(
5         this_bitset_type::operator[]( BitFieldOffset<BitFieldBool>::val ) );
6 }
```

Strongly Typed Declarative Bitsets in C++17

making a new bitset type

- private template alias to the unique namespace for the offset types
- alias for parent bitset type

```
1 // continuation of MAKE_BITFIELDS macro
2 private:
3   using this_bitset_type =
4     std::bitset<BOOST_PP_SEQ_SIZE(
5       BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__))>;
6
7   template<typename T>
8   using BitFieldOffset = BOOST_PP_CAT(name, _detail::BitFieldOffset<T>);
9 }
```

Strongly Typed Declarative Bitsets in C++17

putting it to use

```
1 #include "bitfields.hpp"
2 MAKE_BITFIELDS(Doom, CheckAuthority, ValidateUser, LaunchNukes)
3
4 Doom d { Doom::CheckAuthority{true},
5           Doom::ValidateUser{true},
6           Doom::LaunchNukes{false} };
7
8 if(d.test<LaunchNukes>())
9 {
10   make_war();
11 }
12 else
13 {
14   make_peace();
15 }
```

Strongly Typed Declarative Bits in C++17

[Godbolt compiler explorer](#)

```
1 MAKE_BITFIELD(Testbitset, F1, F2);
2
3 void f1(Testbitset& t1)
4 {
5     t1.set(Testbitset::F1(true),
6            Testbitset::F2(false));
7 }
```

```
1 f1(Testbitset&):
2     mov rax, QWORD PTR [rdi]
3     and rax, -3
4     or rax, 1
5     mov QWORD PTR [rdi], rax
6     ret
```

Strongly Typed Declarative Bits in C++17

[Godbolt compiler explorer](#)

```
1 void f2(std::bitset<2>& t1)
2 {
3     t1.set(0, true);
4     t1.set(1, false);
5 }
```

```
1 f2(std::bitset<2ul>&):
2     mov rax, QWORD PTR [rdi]
3     and rax, -3
4     or rax, 1
5     mov QWORD PTR [rdi], rax
6     ret
```

```
1 f1(Testbitset&):
2     mov rax, QWORD PTR [rdi]
3     and rax, -3
4     or rax, 1
5     mov QWORD PTR [rdi], rax
6     ret
```

Strongly Typed Declarative Bitsets in C++17

summary

- unique details namespace with bitfield offsets and booleans
- type-base access to bits
- bit types exposed as inner types of bitset type
- is still a std::bitset



Can we not do this without preprocessor?

Looking to the future

type generation with metaclasses

- `flag_enum`
- `basic_bool`

Can we not do this without preprocessor?

Looking to the future

flag_enum

- fields that are powers of 2

```
1 flag_enum Doom { auto CheckAuthority, ValidateUser, LaunchNukes; };
2
3 Doom d { Doom::CheckAuthority | Doom::ValidateUser };
4
5 if (d & Doom::LaunchNukes)
6 {
7     make_war();
8 }
9 else
10 {
11     make_peace();
12 }
```

Can we not do this without preprocessor?

Looking to the future

basic_bool

```
1 $class basic_bool : basic_value, comparable {
2     bool val = false;
3
4     explicit basic_bool(bool val): val(val){}
5
6     basic_bool operator& (const basic_bool& other) const { return val & other.val; }
7     explicit operator bool() const { return val; }
8
9     constexpr {
10         compiler.require($basic_bool.variables().empty(),
11                         "basic_bool shouldn't have variables defined");
12         // ...
13     }
14 };
15
16 basic_bool ValidateUser {};      // make new types!
17 basic_bool CheckAuthority {};
```



Strongly Typed Declarative Bitsets in C++17

Strongly Typed Declarative Bitsets in C++17

Questions?