bounded::integer

by David Stone

bounded::integer

- Problems with current approaches
- Attempts to fix the problem
- The bounded::integer solution

The story so far

```
std::numeric_limits<int>::max() + 1
// Undefined behavior
```

The story so far

```
-1 < static_cast<unsigned>(0)

// false
```

The story so far

-1 < static_cast<uint32_t>(0)

// Implementation defined behavior

// Probably false, but true on some Crays

The story so far

sizeof(int) >= 2

// Implementation defined

// Typically 4, but can be as low as 1

The story so far

```
int8_t x = 48;

std::cout << -x;

// Prints -48
```

The story so far

```
int8_t x = 48;

std::cout << +x;

// Prints 48
```

The story so far

```
int8_t x = 48;

std::cout << x;

// Probably prints 0
```

The story so far

```
uint32_t x = 0, y = 1, z = 2;

x < y - z;

// Probably true, depending on sizeof(int)
```

Summary

- Undefined behavior for signed overflow
- Unsigned overflow wraps around
- Comparing signed and unsigned values is confusing
- cstdint types can be characters instead of integers
- integral promotion rules can be unexpected

How To Fix This

Use a bignum

- Fixes overflow issues
- Comparisons work as expected
- Slow

Ban unsigned types

- Fixes mixed-sign comparisons
- Does not fix other issues

CheckedInteger

- Check every operation prior to calculation
- Can fix all issues, but adds overhead everywhere
- Typically throws an exception at run time

Constrained Value

- Proposed for inclusion in Boost
- Much more general than bounded::integer
  - Allows arbitrary restrictions, such as being even or prime or present in a database
- Does not deduce new bounds
- Uses implicit conversions to the underlying type

Ada ranges

- type My_Range is range -3 .. 17;
- Does not deduce new ranges
- Always throws exceptions

bounded::integer

- Replace built-in integers for all use cases
- If there is any overhead at all, the library has failed
  - Don't pay for what you don't use
- Comparisons work as expected
- Enables optimizations
- Enables static analysis

Where to get it

https://bitbucket.org/davidstone/bounded_integer

Supported compilers

- gcc 4.9.0+
- clang 3.4+

Basic usage

```
constexpr bounded::integer<0, 10> x(5);
constexpr bounded::integer<5, 9> y(6);
constexpr auto z = x + y;
// decltype(z) == bounded::integer<5, 19>
std::cout << z << '\n';
// prints 11
```

Policy-driven bounds checking

- bounded::integer<0, 10>
  - Compile-time bounds checking only
- bounded::integer<0, 10, bounded::throw_policy>
  - Run-time bounds-checking via exceptions
- bounded::integer<0, 10, bounded::clamp_policy>
  - Run-time bounds checking with "clamping" or "saturation" behavior

Dynamic bounds checking

- bounded::integer<0, 10, bounded::dynamic_policy<0, 10, bounded::throw_policy>>
  - Has static bounds of [0, 10]
  - Runtime bounds can be narrower
  - Also supports dynamic_min_policy and dynamic_max_policy

Syntax is important

- bounded::integer<0, 10, bounded::dynamic_policy<0, 10, bounded::throw_policy>> is a mouthful
  - Doesn't even fit on one line in this slide!
- bounded::dynamic_integer<0, 10>
  - defaults to throw_policy

Example

```
using namespace bounded;
class Goblin {
public:
    auto heal_self() -> void {
        ++m_health;
    }
    auto take_damage() -> bool {
        --m_health; return m_health == 0;
    }
private:
    dynamic_max_integer<0, 5, clamp_policy> m_health;
};
```

http://strangeguyami.blogspot.com/

How to handle constants?

- Type system does not look at values
- Type of bounded::integer<0, 10> + 5?
  - It's not bounded::integer<5, 15>
- std::numeric_limits<int>
  - Wide bounds, even when constexpr
- bounded::integer<0, 10> + bounded::integer<5, 5>(5) is cumbersome

User defined literal

- Originally not included
- bounded::make<n>() is more general
  - Same as bounded::integer<n, n>(n)

User defined literal

```
auto f() {
    return some_expression + bounded::make<5>();
}
using namespace bounded::literal;
auto g() {
    return some_expression + 5_bi;
}
```

Design decisions

underlying_type

```
enum class storage_type { fast, least };


template<intmax_t min, intmax_t max,
typename overflow, storage_type
storage>

class integer;
```

Inclusive bounds

- "closed range"
- bounded::integer<0, 10>
- std::numeric_limits
- std::uniform_int_distribution

No implicit conversions to int

- bounded::integer never implicitly converts to any built-in type
- Tricky implicit integral promotions
- Implicit narrowing

Conversion to larger type

```
auto f() -> bounded::integer<0, 10> {
    if (something) return 0_bi;
    if (something else) return 6_bi;
    return 10_bi;
}
// Perfectly safe
```

Limitations

Return type deduction

```
auto f() {
    if (something) return 0_bi;
    if (something else) return 6_bi;
    return 10_bi;
}
// error, inconsistent deduction for 'auto'
```

Conditional statements

- b ? 1_bi : 2_bi; // fails to compile
- BOUNDED_CONDITIONAL(b, 1_bi, 2_bi);
  - Oh no! Not a macro!
- has type bounded::integer<1, 2>
- std::common_type defined in terms of ?:
  - Should be the other way around

Non-type template parameters

- literal class types cannot be non-type template parameters
- template<integer<0, 9> x>
  - illegal

range limited to intmax_t

- Limits some values
  - Large values of uintmax_t
  - Any floating point

bounded::integer

- https://bitbucket.org/davidstone/bounded_integer
- david@doublewise.net

Bonus slide: array

```
using index_t = bounded::checked_integer<0, size - 1>;
auto operator[](index_t index) -> T & {
    return m_array[index.value()];
}
template<typename Index>
auto at(Index index) -> T & {
    return m_array[static_cast<index_t>(index).value];
}
```