Integer Type Selection

in Safe, Secure, Correct Code

Robert C. Seacord

Standardization Lead

Robert.Seacord@woven.toyota

Use strong typedefs


Fin

Agenda

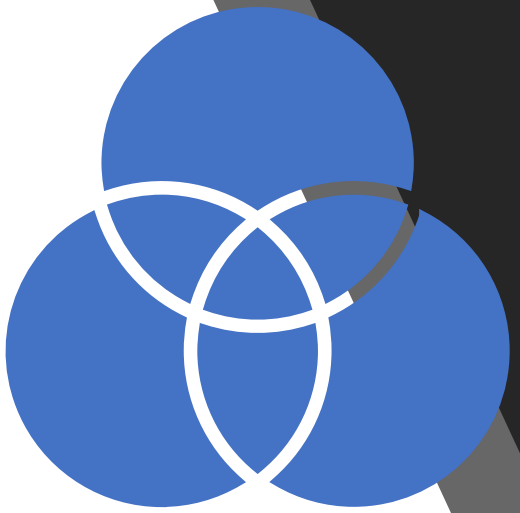
- Integers basics
- Signedness
- Exact width types

Signed and Unsigned Types

There are five standard signed integer types: `signed char`, `short int`, `int`, `long int`, and `long long int`.

For each of the standard signed integer types, there exists a corresponding (but different) standard unsigned integer type.

Signed Integers



Signed integers are used to represent positive and negative values, the range of which depends on the number of bits allocated to the type and the representation.

Each unsigned integer type has a corresponding signed integer type which occupies the same amount of storage.

Representing Negative Values

All sufficiently small non-negative values have the same representation in corresponding signed and unsigned types.

The sign bit is treated as the highest-order bit and indicates whether the represented value is negative.

Up through C17 and C++17, negative values can be represented as

- sign and magnitude
- ones' complement
- two's complement

Starting with C23 and C++20 only two's complement is supported.

Two’s Complement

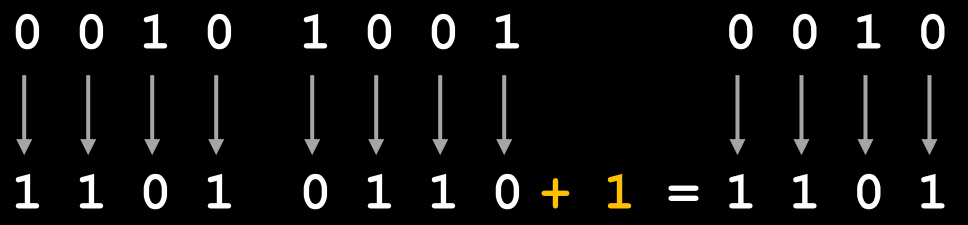
The sign bit is given the weight $-(2^{N-1})$, and the other value bits have the same weights as for unsigned.

For example, **1111010101** equals −43 in two's complement representation.

Given a width of 10 bits, the sign bit is given the weight $-(2^9)$ or −512.

The remaining bits equals 469, so 469−512 = −43.

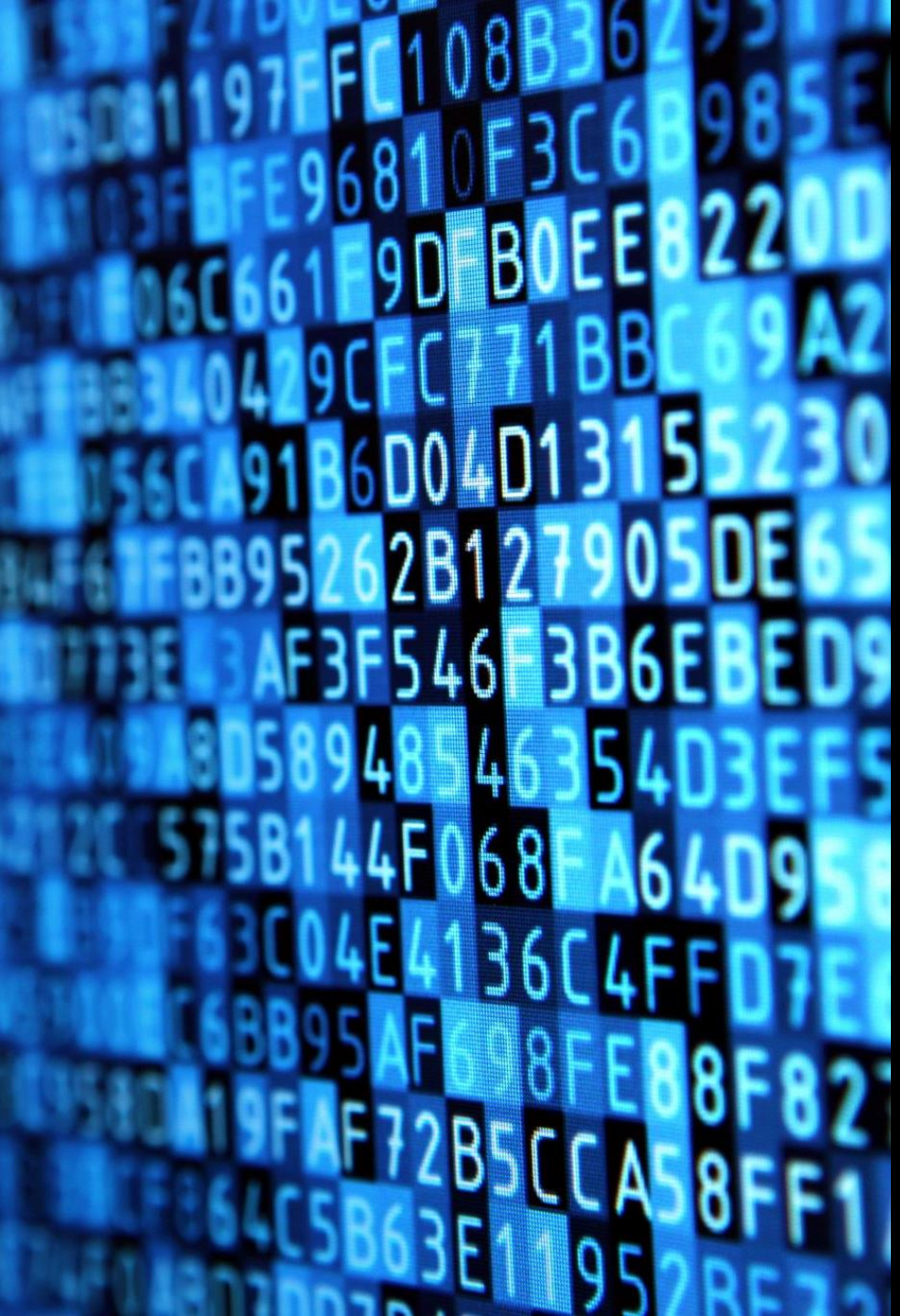
To negate a two’s complement value, first form the one's complement negation and then add 1 (with carries as required).



8-bit Two's Complement Values

Binary	Decimal	Weighting	Constant
00000000	0	0	
00000001	1	2^0	
01111110	126	$2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1$	
01111111	127	$2^{N-1} - 1$	SCHAR_MAX
10000000	−128	$-(2^{N-1})+0$	SCHAR_MIN
10000001	−127	$-(2^{N-1})+1$	
11111110	−2	$-(2^{N-1})+126$	
11111111	−1	$-(2^{N-1})+127$	

Binary and decimal representations for an 8-bit two's complement (signed) integer type with no padding (that is, N=8).



Unsigned Integers

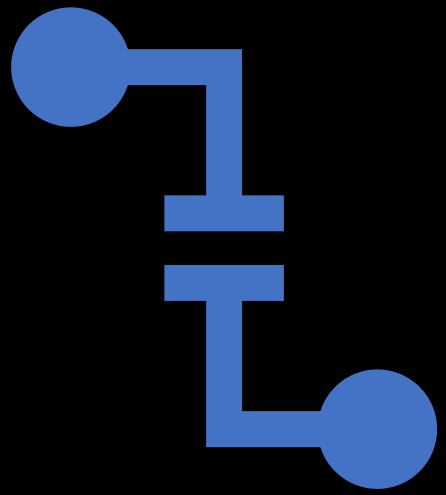
Unsigned integer types represent values using a pure binary system with no offset.

- The right-most bit has the weight 2^0
- The next bit to the left has the weight 2^1 and so forth

The value is the sum of all the set bits.

Unsigned integer values range from 0 to a max that depends on the width of the type.

This maximum value can be calculated as 2^N-1, where *N* is the width.



Unsigned Integer Uses

All bitwise operators (`|` `&` `^` `~`) treat the bits as pure binary model, e.g.:

```
  1 1 0 1 = 13
^ 0 1 1 0 =  6
---------------
= 1 0 1 1 = 11
```

Overflow and Wraparound

overflow - portion of a numeric word expressing the result of an arithmetic operation by which its word length exceeds the word length provided for the number representation [ISO/IEC 2382:2015]

wraparound - the process by which a value is reduced modulo 2^N where N is the width of the resulting type

Only signed integer operations can overflow

A computation involving unsigned operands can never overflow, because arithmetic for the unsigned type is performed modulo 2^N.

Signed/Unsigned Optimizations

Multiply a signed/unsigned value by 7 and divide by 7 again:

```
volatile int si1 = INT_MAX;
volatile unsigned int ui1 = INT_MAX;
printf("si1 * 7 / 7 = %d\n", si1 * 7 / 7);
printf("ui1 * 7 / 7 = %u\n", ui1 * 7 / 7);
```

Has UB

Modulo behavior

Output:

si1 * 7 / 7 = 2147483647 ✓

ui1 * 7 / 7 = 306783377 ✗

Signed Operations

Signed operation is optimized because of the undefined behavior

Produces the correct mathematical result

```
mov     edi  OFFSET FLAT .LC0
xor     eax  eax
mov     DWORD PTR  rsp 8   2147483647
mov     DWORD PTR  rsp 12   2147483647
mov     esi  DWORD PTR  rsp 8
call    printf
```

Unsigned Operations

Unsigned operation is unoptimized because of the compiler is required to wrap

Produces the incorrect mathematical result

```
mov     edx  DWORD PTR  rsp 12
mov     edi  OFFSET FLAT .LC1
lea     eax   0 rdx 8
sub     eax  edx
mov     edx  eax
imul    rdx  rdx  613566757
shr     rdx  32
```

```
sub     eax  edx
mov     esi  eax
xor     eax  eax
shr     esi
add     esi  edx
shr     esi  2
call    printf
```

All std containers use size_t as the size_type

size_t

size_t is the unsigned integer type of the result of the sizeof operator and is defined in <stddef.h> or <cstddef>

Variables of type size_t are guaranteed to be of sufficient precision to represent the size of an object.

The limit of size_t is specified by the SIZE_MAX macro.

K&R C didn't provide size_t.

size_t was introduced by C89 to eliminate a portability problem because

- unsigned int may be too small to represent the size of the address space
- unsigned long long may be too large (and consequently inefficient)

`size_t` Use Cases

The portable and efficient way to declare a variable containing a size is

```
size_t n = sizeof(thing);
```

Or to define a function `foo` that takes a size argument is

```
void foo(size_t size);
```

`size_t` is a good type to use for local variables within these functions count up to or down from `size` and index into arrays.

Similarly, variables that represent the count of elements in an array should be declared as `size_t`, particularly for character arrays where the element count can be as large as the largest object that can be allocated in the system.

Template	Comments
`template<class T> struct make_signed;`	If `T` names a (possibly cv-qualified) signed integer type then the member typedef `type` names the type `T`; otherwise, if `T` names a (possibly cv-qualified) unsigned integer type then `type` names the corresponding signed integer type, with the same cv-qualifiers as `T`; otherwise, `type` names the signed integer type with smallest rank for which `sizeof(T) == sizeof(type)`, with the same cv-qualifiers as `T`. *Mandates*: `T` is an integral or enumeration type other than *cv* `bool`.
`template<class T> struct make_unsigned;`	If `T` names a (possibly cv-qualified) unsigned integer type then the member typedef `type` names the type `T`; otherwise, if `T` names a (possibly cv-qualified) signed integer type then `type` names the corresponding unsigned integer type, with the same cv-qualifiers as `T`; otherwise, `type` names the unsigned integer type with smallest rank for which `sizeof(T) == sizeof(type)`, with the same cv-qualifiers as `T`. *Mandates*: `T` is an integral or enumeration type other than *cv* `bool`.

`ssize_t`

Defined by POSIX.1-2017

`ssize_t` is used for functions whose return value could either be a valid size, or a negative value to indicate an error.

`ssize_t` shall be a signed integer type

The type `ssize_t` shall be capable of storing values at least in the range `[-1, {SSIZE_MAX}]`.

In-band error indicators not supported by the language (such as a null pointer) are generally a bad idea.

Agenda

- Integer basics
- Signedness
- Exact width types

Poll Question

Do you only use unsigned integers for modulo behavior?

Do you only use unsigned integers to represent values that cannot be negative?

I'm only here for the snacks.

Loop Limited by Lower Bound

An unsigned-integer expression can never evaluate to `< 0` because of wraparound.

It is possible to code tests that are always `true` or always `false`.

Because `i` cannot take on a negative value, this loop never terminates:

```
for (size_t i = size; i >= 0; --i)
```

Such tests are probably coding errors, but this wraparound-induced infinite loop is not considered to be an error according to the C or C++ standards.

Using a Signed Integer

Can this be improved by using a signed integer?

```
for (ssize_t i = (ssize_t)size-1; i >= 0; i--)
```

The loop terminates.

Signed integers have normal behavior around zero, a common value.

However, this code contains an unsigned to signed type conversion.

Not all values of the original type can be represented in the resulting type.

Conversion to Signed Integer

A value for `size` that cannot be represented as `ssize_t` is converted to a negative signed value causing the immediate termination of the loop.

```
for (ssize_t i = (ssize_t)size-1; i >= 0; i--)
```

Consequently, this approach is defective.

Unsigned Integer with Wraparound

Another approach is to initialize `i` to `size - 1` and decrement on each iteration.

When the counter reaches zero, the decrement causes the counter to wraparound to the maximum possible value of the unsigned type (`SIZE_MAX`).

This value of `i` is now larger than `size`, so `i < size` evaluates to `false`, and the loop terminates.

```
for (size_t i = size - 1; i < size; i--)
```

Wraparound is well defined in C and C++.

Diagnosed by `-fsanitize=unsigned-integer-overflow`

`do-while` loop

Use a **`do-while`** loop that tests the condition at the end of the loop

```
size_t i = size;
do {
  i--;
  …
} while (i != 0);
```

Eliminates wraparound and potential diagnostic.

Unsigned Integer Problems

```
if (endIndex - startIndex >= safety_margin) {
  // use fast algorithm with unchecked access
}
```

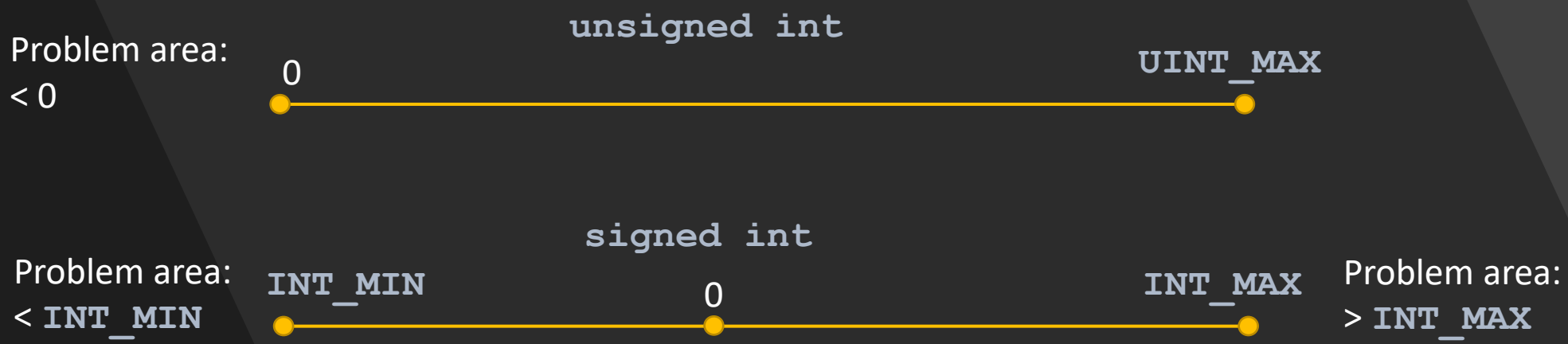
`safety_margin` is some reasonably small integer.

If the programmer fails to eliminate the `endIndex < startIndex` edge case, the check passes because of wraparound.

Problem Areas

Both unsigned and signed operations can be erroneous.

With unsigned, the problem area is nearby common use cases.



Untrusted inputs can cause UB beyond signed integer boundaries.

Observations

If developers are using signed integers to avoid thinking of the behavior around zero, they are probably not thinking about overflow behavior.

Security- and safety-critical systems cannot tolerate sloppy thinking of this kind.

Infinite loops are easily detected during testing.

Signed Integers are Faster

Part of this misconception about performance stems from a talk by Chandler Carruth at CppCon 2016
https://youtu.be/yG1OZ69H_-o

64-bit Architecture Code Example

```
int f(uint32_t i1, uint32_t i2, uint8_t *block) {
    uint8_t c1, c2;
    c1 = block[i1];
    c2 = block[i2];
    if (c1 != c2) return (c1 > c2);
    i1++; i2++;
    // repeat ad nauseum
    return 0;
}
```

64-bit operation

Chandler argues that this code is slower using unsigned integers because it is required to wrap while signed integer overflow has undefined behavior and can be ignored.

Reality

```
  mov     al, byte ptr [rdx + rsi]
  cmp     byte ptr [rdx + rdi], al
  jne     .LBB2_2
  mov     al, byte ptr [rdi + rdx + 1]
  cmp     al, byte ptr [rsi + rdx + 1]
.LBB2_2:
  seta    al
  movzx   eax, al
  ret
```

On 64-bit architectures, `size_t` is a 64-bit unsigned integer.

The `size_t` type produces the fastest code at `-O3` on gcc, icc, and clang.

The relative performance `int32_t` and `uint32_t` depends on the compiler, but is always worse than `size_t`

Properly typed (unsigned) integers produce the fastest code.

Chandler Carruth @chandlerc1024 · Apr 29

Replying to @RCS and @lefticus

Just to be clear, this is a 6 year old talk... Compilers have actually changed in this specific area significantly.

This is also an incredibly brittle area. In retrospect, I should have used a more durable example. And the idea was never to *introduce* UB, but that got lost...

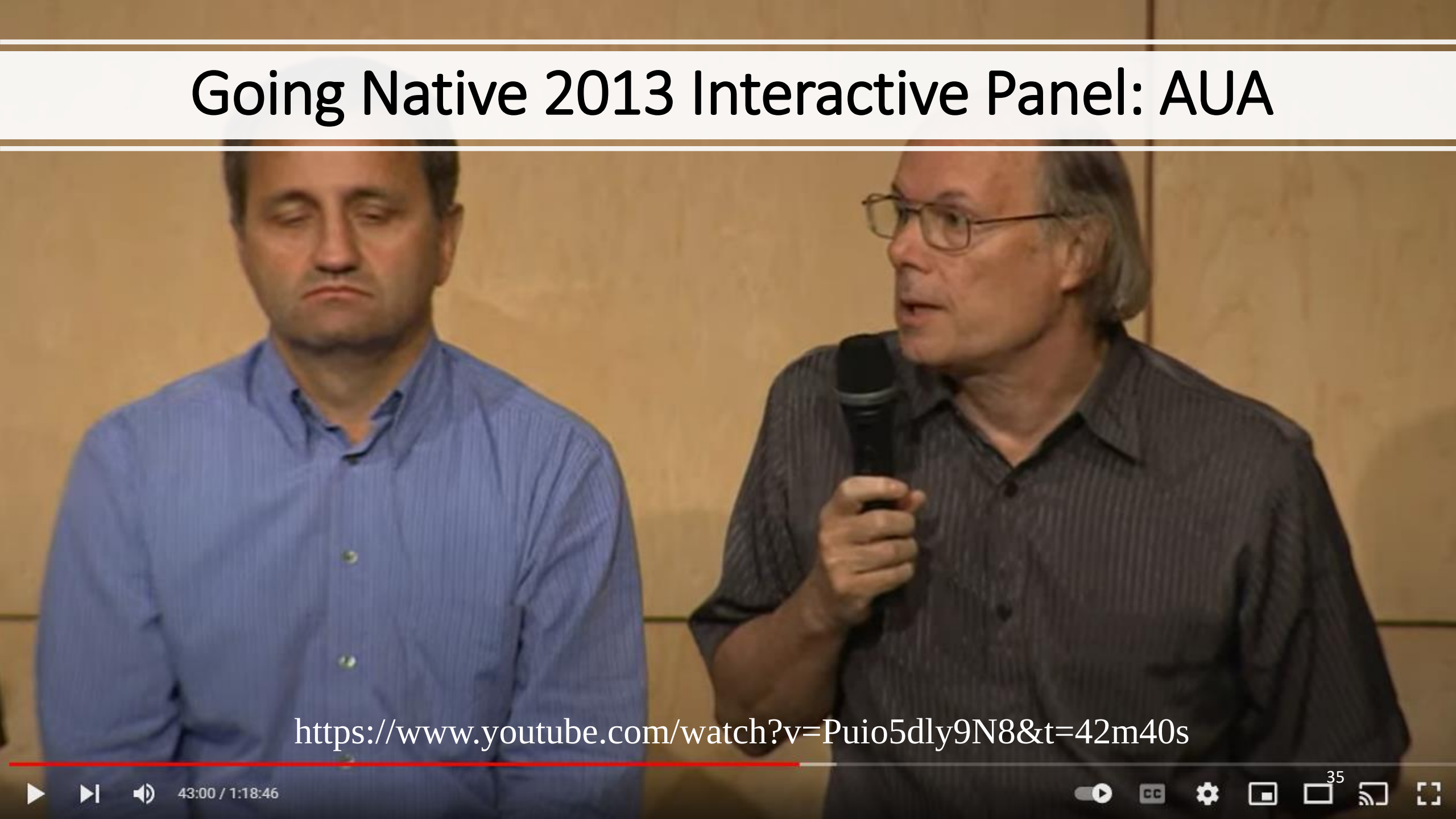
1 7

Chandler Carruth @chandlerc1024 · Apr 29

I largely don't think the point I was hoping to make came across well, and somewhat wish folks would not cite this talk at all given that.

2 6

Going Native 2013 Interactive Panel: AUA



Google C++ Style Guide (1 of 2)

Unsigned integers are good for representing bitfields and modular arithmetic.

Because of historical accident, the C++ standard also uses unsigned integers to represent the size of containers—many members of the standards body believe this to be a mistake, but it is effectively impossible to fix at this point.

The fact that unsigned arithmetic doesn't model the behavior of a simple integer but is instead defined by the standard to model modular arithmetic (wrapping around on overflow/underflow), means that a significant class of bugs cannot be diagnosed by the compiler.

In other cases, the defined behavior impedes optimization.

Google C++ Style Guide (2 of 2)

That said, mixing signedness of integer types is responsible for an equally large class of problems.

The best advice we can provide: try to use iterators and containers rather than pointers and sizes, try not to mix signedness, and try to avoid unsigned types (except for representing bitfields or modular arithmetic).

Do not use an unsigned type merely to assert that a variable is non-negative.

UndefinedBehaviorSanitizer (UBSan)

Many vulnerabilities are discovered using UBSan + fuzzing.

`-fsanitize=signed-integer-overflow` diagnoses signed integer overflow, where the result of a signed integer computation cannot be represented in its type.

Includes checks covered by

- `-ftrapv`
- signed division overflow (`INT_MIN/-1`)

Does not include checks for lossy implicit conversions performed before the computation.

-fsanitize=unsigned-integer-overflow

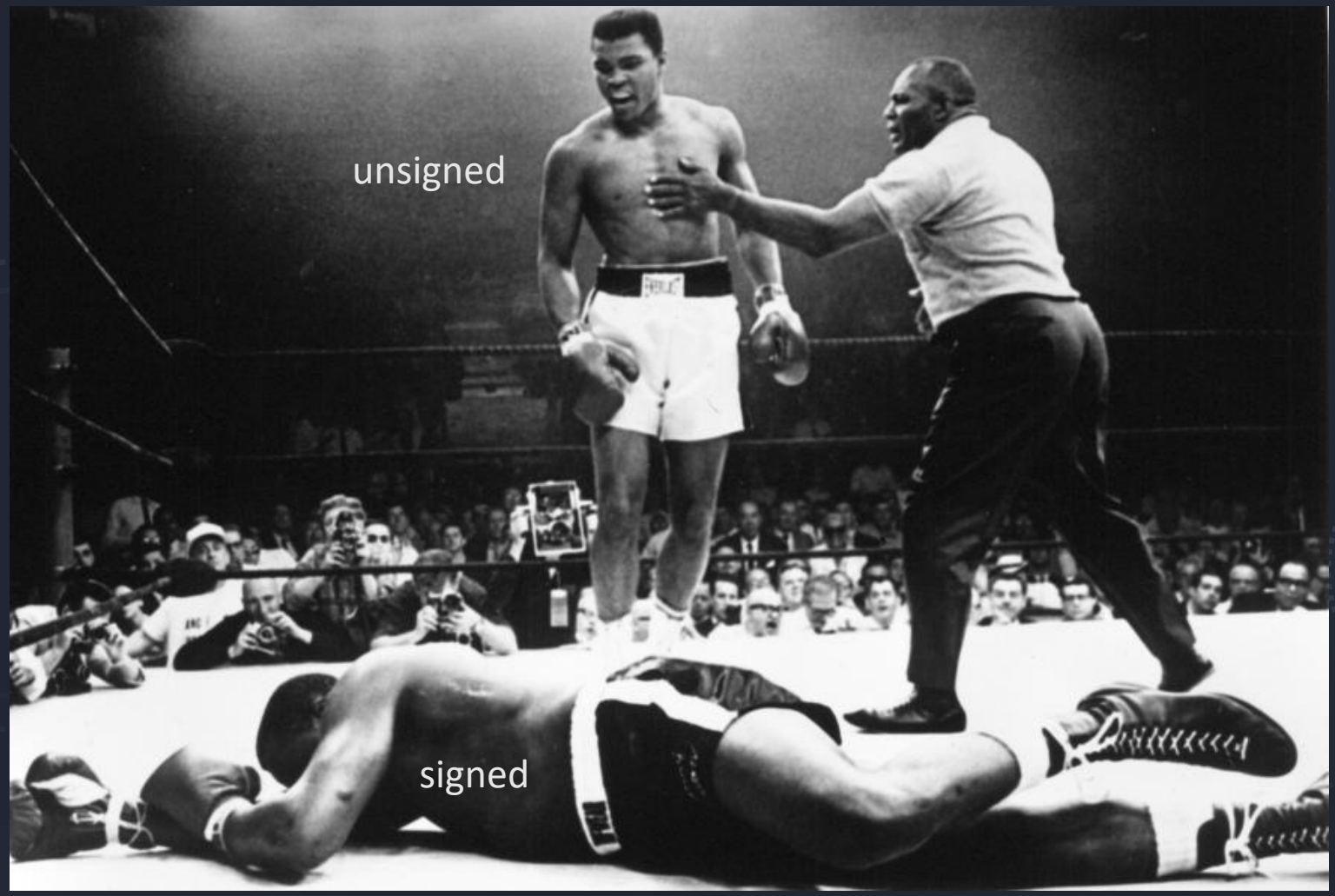
```
uint32_t h(uint32_t i)
{
  return i - 100;
}
```

```
h:                                      # @h
        push    rbx
        mov     ebx, edi
        sub     ebx, 100
        jb      .LBB2_1
        mov     eax, ebx
        pop     rbx
        ret
.LBB2_1:
        mov     esi, edi
        mov     edi, offset .L__unnamed_3
        mov     edx, 100
        call    __ubsan_handle_sub_overflow
        mov     eax, ebx
        pop     rbx
        ret
```

/app/example.c:31:14: runtime error: unsigned integer overflow: 90 - 100 cannot be represented in type 'unsigned int'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior /app/example.c:31:14 in

Lesson

Signed integers are the same or worse than unsigned integers for division and remainder operations.



Operators that Wraparound

Op	Wrap	Op	Wrap	Op	Wrap
+	✓	*=	✓	&	
-	✓	/=		|	
*	✓	%=		^	
/		<<=	✓	~	
%		>>=		!	
++	✓	&=		un +	
--	✓	|=		un -	✓
=		^=		<	
+=	✓	<<	✓	>	
-=	✓	>>			

Operators That Can Overflow

Op	Overflow	Op	Overflow	Op	Overflow
+	✓	*=	✓	&	
-	✓	/=	✓	|	
*	✓	%=	✓	^	
/	✓	<<=	✓	~	
%	✓	>>=		!	
++	✓	&=		un +	
--	✓	|=		un -	✓
=		^=		<	
+=	✓	<<	✓	>	
-=	✓	>>			

Integer Division

On x86, a signed integer overflow condition occurs when the minimum integer value for a 32-bit or 64-bit integer is divided by -1.

The x86 division `idiv` instruction does not set the overflow flag.

A division error is generated if

- the source operand (divisor) is zero
- the quotient is too large for the designated register

A divide error results in a fault on interrupt vector 0.

Remainder

The result of the `/` operator is the quotient from the division of the first operand by the second; the result of the `%` operator is the remainder.

In both operations, if the value of the 2nd operand is zero, the behavior is undefined.

When integers are divided, the result of the `/` operator is the algebraic quotient with any fractional part discarded.

If the quotient `a/b` is representable, the expression `(a/b)*b + a%b` shall equal `a`.

`INT_MIN % -1`

Many CPUs implement remainder as part of the division operator, which can overflow.

Overflow can occur during a remainder operation when the dividend is equal to the minimum (negative) value for the signed integer type and the divisor is equal to -1.

This occurs in even though the mathematical result of such a remainder operation is 0.

Lesson

Easier and faster to check for unsigned wraparound than signed overflow

Signed vs. Unsigned Summary

Errors can occur using both signed and unsigned integers.

Signed integers have more ways to fail.

Injecting UB into a program does not make it safer or more secure.

Unsigned integers are less expensive to program safely & securely.

A common misconception among WG21 Committee Members is that signed integers should be the default integer type. The opposite is true; unsigned integers should be the default type for representing integers that cannot have negative values.

Agenda

- Integer basics
- Signedness
- Exact width types

AUTOSAR

Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 19-03

Rule A3-9-1

Rule A3-9-1 (required, implementation, automated)

Fixed width integer types from `<cstdint>`, indicating the size and signedness, shall be used in place of the basic numerical types.

Standard types such as `char`, `int`, `short`, `long` should not be used.

MISRA C++ 2008 has a similar rule (Rule 3-9-2).

```
"Error: restd::int32_terpret_cast not found."
```

Exact-width integer types

`intN_t` is a signed integer type with width `N` and no padding bits.

`int8_t` denotes such a signed integer type with a width of exactly 8 bits.

`uintN_t` designates an unsigned integer type with width `N` and no padding bits.

`uint24_t` denotes such an unsigned integer type with a width of exactly 24 bits.

If an implementation provides standard or extended integer types with a particular width and no padding bits, it must also define the corresponding typedef names.

These are optional types—targets that lack the corresponding word size won't implement these types.

Minimum-width integer types

`int_leastN_t` is a signed integer type with a width of at least `N`.

No signed integer type with lesser size has at least the specified width.

`int_least32_t` is a signed integer type with a width of at least 32 bits.

`uint_leastN_t` is an unsigned integer type with a width of at least `N`.

No unsigned integer type with lesser size has at least the specified width.

`uint_least16_t` is an unsigned integer type with a width of at least 16 bits.

`uint_least16_t` has the same basic properties as `unsigned short`—they both guarantee a minimum range but may be larger.

Fastest minimum-width integer types

`int_fastN_t` is the fastest signed integer type with a width of at least `N`.

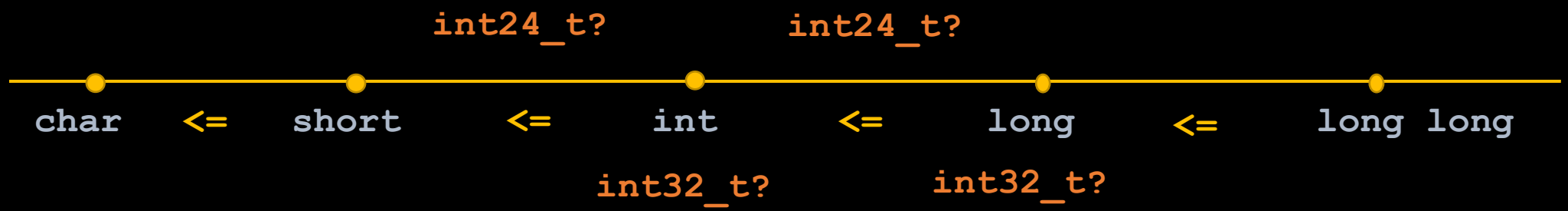
`uint_fastN_t` is the fastest unsigned integer type with a width of at least `N`.

Exact Width Types are Detached from the Standard

`int` is the natural width suggested by the architecture

A prvalue of an integer type other [...] whose integer conversion rank is less than the rank of `int` can be converted to a prvalue of type `int` if `int` can represent all the values of the source type; otherwise, the source prvalue can be converted to a prvalue of type `unsigned int`.

The standard signed integer types have minimum ranges and “each type provides at least as much storage as those preceding it”.



Matching APIs

Integer type selection is largely driven by the need to match APIs.

If the API accepts an `int` parameter and you pass an `int32_t`, how will your code behave on if `int` is implementation defined to be not 32 bits (e.g., 16)?

Using a different type requires checking value ranges before each call.

Converting between implementation-defined types and exact-width types is difficult and error prone.

Corollary: Extremely important to select appropriate integer types in API design

Matching APIs

Somewhat feasible for user-defined code.

Problematic when interacting with standard library code.

Matching APIs – Return Values

```
int i = strcmp("", "");  // non-compliant
auto i = strcmp("", ""); // not diagnosed?
```

In practice, catching an `int` return value from a function with an `auto` variable does not trigger a violation of this rule with most tools, which is a good way to violate the rule without getting caught.

Now we are writing bad code, to violate a bad rule, to avoid detection.

How is this rule helping us again?

Integer Type Selection

Exact-width types are useful when representing fixed width registers or fields in network protocols or file formats, particularly when reading into a `struct`.

Minimum width integer types can be portably used to ensure you get the smallest type of some size.

Fastest minimum-width integer types guarantee a minimum range but optimize for performance over storage.

Rule A3-9-1 `size_t`, `ptrdiff_t`

Rule A3-9-1 is silent on the use of `size_t` and `ptrdiff_t` but as these are not fixed width types the assumption is that they should not be used.

Requires reproducing the mechanism in a suboptimal manner:

```
#include <mylib.h>
using my_size_type = // uint32_t or uint64_t;
void *my_malloc(my_size_type size);
int main() {
    my_size_type size = large_number;
    void *p = my_malloc(size);
}
```

Non-Insane Solution

Do not diagnose the use of `size_t` and `ptrdiff_t`.

Use static assertions to ensure that integer sizes meet your minimum requirements.

Thanks!

Robert C. Seacord
Standardization Lead
Twitter: @rcs
LinkedIn.com/in/robertseacord/
T: +1 (412) 580-2981
E: Robert.Seacord@woven.toyota
W: woven.toyota