C programming techniques and Neovim-specific guidance

sizeof

`sizeof(<var>)` vs `sizeof(<type>)`

> *something you have to be really careful about: the difference between arrays and pointers [...] be judicious: if the variable is simple (int, long, ...), use `sizeof(variable)`, if the variable is complex (struct, pointer-to-pointer, ...), use `sizeof(the_actual_type)`.*

Scope

It is **undefined behavior** to access a pointer that was assigned in an inner scope.

Struct organization

https://github.com/neovim/neovim/pull/656#issuecomment-41905534

TODO: link to discussion of legacy Vim struct hack

Unsigned or signed? Integer overflow/underflow

- Conversion of signed variables to unsigned (in files not checked by `-Wconversion`)

There are a few very important things to keep in mind while choosing between signed and unsigned integral types:

Firstly, **unsigned overflow is defined**, while **signed overflow is not**. This is an unfortunate historical oversight stemming from a time where it wasn't sure what representation a signed integer would have. The C standard decided that it shouldn't/couldn't specify what would happen when a signed integer overflows, because each representation would have different behaviour. In modern times, signed integers are always represented in two's complement form, which has many advantages.

> *An interesting thing to note is that it is possible to force gcc and clang to view signed overflow as defined (wraparound, like unsigned), by passing the -fwrapv flag. This is, unfortunately, non-standard.*

What does this mean, defined vs. undefined? If an unsigned integer overflows, it wraps around back to zero (it's modulo addition). Yet, if a signed integer overflows, $deity only knows what will happen. More specifically: we *know* what would happen if a two's complement signed integer would overflow, but the compiler can do whatever it wants, because the standard says it is undefined. As a consequence, an optimizing compiler will often assume that a signed integer *cannot* overflow and optimize out some if-branches or comparisons. This behavior can cause loops to run forever. Note that if the conditions are not written carefully, even the well-defined wraparound overflow of unsigned integers can cause non-terminating loops, `(U)INT_MAX/MIN` are your friends.

Thus it would seem that unsigned arithmetic is superior, because it has defined over- and underflow. But that's not always true. There's a good reason why many languages (like Java) don't expose unsigned types: they can cause difficult to spot errors. The most common form of under/overflow is **underflow in unsigned arithmetic**. Subtracting 1 from `unsigned int num = 0;` will make it wrap around to `UINT_MAX`. This is much more common than one would think. For this reason alone, it is usually **much** safer to use a plain `int` as a loop counter instead of `uint32_t` / `size_t` /... or another unsigned type. Even seasoned programmers find it difficult to avoid writing unsigned code that doesn't underflow in some cases.

Problem: correct signed code is easier to write, but you have to use casts when comparing to `size_t` (which happens often, as it is the return type of `sizeof`, `strlen` and many others). Casts are ugly and should be avoided if at all possible. But we cannot avoid them everywhere. Sometimes, a trade-off has to be made. See [previous `-Wconversion` PRs] for examples.

Conclusion:

- if there is any chance of underflow or the loop in question is small (definitely less than `2^31` items), use signed arithmetic and a guard before the loop.
- if there is any chance of overflow, use unsigned arithmetic and possibly guards.
- if there is a chance of both underflow and overflow, be extremely careful and paranoid (guards/asserts).

Guarded casting

Fixed-size vs. generic types

Should we use `(u)intX_t` and friends over `char`, `short`, `int`, `long` et al.? ...

- `size_t`: ...
- `rsize_t`: this type is new in the C11 standard, which is why we can't use it, but the reasoning and usage behind it are interesting. Instead of `RSIZE_MAX` being the actual maximum value that a variable of type `rsize_t` can have, it is **less** than that. This means that one can check if `val <= RSIZE_MAX` before continuing operations and have it be useful. The first useful property is: values about `RSIZE_MAX` are usually too large to be useful anyway. Who wants to allocate such titanic amounts of memory anyway, if `rsize_t` is 64-bits? Arguably it would be better to refuse to perform the operation. The second useful property is nice to make unsigned arithmetic less susceptible to the dreaded underflow problem. `rsize_t val = -4` will be larger than `RSIZE_MAX` because it has wrapped around. Any functions that does a bounds-check on `RSIZE_MAX` will reject that value. This seems to be a good way to interact with unsigned code without needing to cast (`rsize_t` should be unsigned) and still have the safety advantages of signed arithmetic.

Tools and articles

- Secure C Coding
- Modern source-to-source transformation with Clang and libTooling
- How Should You Write a Fast Integer Overflow Check?
- Stubborn and ignorant use of int where size_t is needed

Undefined behavior

- https://cryptoservices.github.io/fde/2018/11/30/undefined-behavior.html