Introduction

In many different tasks, the need arises to refactor integer types, mainly to:

- avoid errors because of mixing of signed/unsigned types and implicit conversions.
- emphasize size semantics where appropriate.
- conform with system functions returning `size_t` values without the need of casting.
- ensure no unneeded limits are imposed on sizes of things we can operate on.
- avoid possible data truncation because of explicit casting.
- etc.

It's important to agree on how this kind of refactoring should be done. However:

- This is a difficult thing to get completely right. No hard-and-fast set of rules will match every possible situation.
- **This set of guidelines intends to be a framework establishing principles** to help new people to confront these problems, and for more experienced people to have a consensus so that code is homogeneous.
- **It does not intend to be a set of mechanical rules to be applied without further thinking, or to be dogmatic about**.
- Every particular case should be carefully analyzed before taking actions on it.

That said, here goes the advice:

Long types

- `long_u` with size semantics (all, or mostly all) --> `size_t`
- `long_u` without size semantics (rare, if ever) --> `uint64_t`
- `long` with size semantics:
  - signedness conversion easy --> `size_t`
    (check for signedness conversion usual problems).
  - signedness conversion difficult --> `ssize`
    (for example, complicated code involving subtractions)
    Note: `ssize` is a project-wide typedef defined in `src/nvim/types.h`.
- `long` without size semantics --> `int64_t`

Int types

- `int` with size semantics:
  - signedness conversion easy --> `size_t`
    (check for signedness conversion usual problems).
  - signedness conversion difficult --> `ssize`
    (for example, complicated code involving subtractions)
- `int` without size semantics --> `int`

Special cases

In spite of the general advice above, there are some cases where we prefer more tight typing.

Struct fields

We should try to keep structs small, if possible. To that end:

- Use fixed width types (`int32_t` , `uint32_t` , etc.), if possible. This is:
- Do that only if you can be sure that the specified width will always be enough. So:
- Please don't impose arbitrary/unneeded limits on fields.

- For example: If a field has clear size semantics and there's no particular reason to impose a restriction on it, use `size_t` / `ssize`.

External interfaces

- For functions interfacing other processes over a transport/serialization mechanism, fixed-width types are preferred. For example, in `msgpack_rpc.h`:

  ```c

bool msgpack_rpc_integer_result(uint32_t result, msgpack_object *req, msgpack_packer *res);

````
- For functions part of a public API, native types are preferred. For example, in a hypothetical `libneovim.h`:

  ```c
int neovim_get_current_buffer(void);
````

Type cascading

Once you have some input variable you have to deal with (be it a struct field, a function parameter, or even a global), the issue arises whether to cascade that type in code dealing with it, or using a wider type if considered better for some reason. Typical example is, once you have a fixed-width struct field, code dealing with it (function variables/parameters) should also use fixed-width types, or could types we widened to some other enclosing type? In principle, we say:

- If access to input variable is read-only, then it's safe to use wider types if some other reason makes them preferable, as you will be only upcasting the value.
- If access to input variable can be read/write, then intermediate variables/params should try to maintain input variable's type as much as possible. If not, you will end up with a downcast somewhere that will require an assert/some other kind of guard/error handling.

Loops

We find [this]() pretty convincing, taking great care not to incurr in errors described [here](). From that, we conclude:

In loops, we have a *counter* variable and a *limit* expression (*condition* being a comparison between *counter* and *limit*). Issues can arise mainly because of implicit conversions in *limit* expression, as well as mixing different-signedness types in *condition* (i.e., when *counter* and *limit* have types of different signedness). To avoid/reduce implicit conversion and type-signedness-mixing problems:

- If possible, try to avoid different-signedness types in variables within *limit* expression (many errors are because implicit conversion from signed type to unsigned one).
- In principle, *limit* expression's type determines *counter*'s type. If *limit* expression is `size_t`, so is counter. If *limit* expression is `ssize`, so is counter. And so on.
- If resulting type of *counter* and *limit* is unsigned:
  - Check *limit* expression for frontier values (e.g., when size is zero).
  - Avoid *condition* using substractions (unless guarded so that it can be proved for result to always be positive). Prefer equivalent condition using additions on the other side.
- As an optimization, you could use plain `int` instead of `ssize`, or `unsigned int` instead of `size_t`, but only if you are sure that those types will be enough always. Please try not to impose arbitrary/unneeded limits.