Neovim is a big refactoring effort. Bringing ideas from the literature on this subject should help. Contributors should be encouraged to detect bad code smells and apply the necessary refactorings with the caution that's necessary when dealing with old/undocumented/untested code.

The lists here are not exhaustive and should not be seen as a perfect prescription for the problems that may be found in code. Be flexible.

Catalog of C Refactorings

A catalog of C refactorings that may guide us while improving neovim's codebase.

From Garrido 2000 **2.1 Proposal for a List of C Refactorings** (page 17)

Adding a Program Entity

- **Add a variable**
- **Add a parameter to a function**
- **Add a typedef definition encapsulating an existing type**
- **Add a field to a structure**
- **Add a pointer to a variable:** add a declaration of a pointer variable with pointed type equal to the type of the selected variable, and assign to the pointer the address of the selected variable.

Deleting a Program Entity

- **Delete unused variable**
- **Delete unused parameter**
- **Delete a function**

Changing a Program Entity

- **Rename variable**
- **Rename constant**
- **Rename user-defined type**
- **Rename structure field**
- **Rename function**
- **Replace the type of a program entity**
- **Contract variable scope**
- **Extend variable scope**
- **Replace value with constant**
- **Replace expression with variable**
- **Convert variable to pointer**
- **Convert pointer to direct variable access**
- **Convert global variable into parameter:** find all functions that access the given global variable. For each of the functions, add a formal parameter to the function definition, add the global variable as an actual parameter in the function call, and replace access to the global variable inside the

function for the parameter. The user may choose if the parameter should be a pointer type.
- **Reorder function arguments:** reorder the arguments in a function definition and in all calls to that function.

Complex refactorings

- **Group a set of variables in a new structure:** (e.g., #775) legacy systems often overuse global variables, which make a program non-reusable. Most changes, as minor as they can be, require global update. However, programmers use global variables when they otherwise should pass many of them as parameters. Passing too many parameters to a function can increase the calling time.

  The remedy to this problem is to define data structures grouping isolated variables, and pass a single reference to the structure as parameter. This refactoring defines structures by grouping existing variables. The second part of converting global access to parameters by reference is handled by the refactorings "Convert global variable into parameter".

- **Extract function**: ~~replace a complicated expression or statement list with a function call. As pointed out in [Fowler 99], this is a very common transformation. When a function gets too long or complicated, a good practice is to divide it into smaller fragments, turn each fragment into a function and replace the long function for a shorter one with function calls. The code extracted is scanned and for each reference to local variables in the source function, a parameter is added to the extracted function.~~ I (@philix) no longer recommend function extraction as a good refactoring pattern. I think one should only extract functions when (1) the new function can be pure (https://en.wikipedia.org/wiki/Pure_function) or (2)there's already a concrete case of reuse. John Carmack discusses this topic on John Carmack on Inlined Code.

- **Inline function**

- **Consolidate conditional expression**: join adjacent cases in switch.

- **For into while**

- **While into for**

- **While into do while:** transform the statement, reversing the condition of the while for the do while.

Code Smells

List of code smells by Jeff Atwood that may indicate solvable problems and inspire neovim contributors. This list was made for object oriented programming, but it's not hard to see how most of them apply to C.

Code Smells Within Classes (.c files)

- **Comments**
- **Long Method (long functions)**
- **Long Parameter List**
- **Duplicated Code**
- **Conditional Complexity**
- **Combinatorial Explosion**
- **Large Class (large .c files)**
- **Type Embedded in Name**
- **Uncommunicative Name**
- **Inconsistent Names**
- **Dead Code**
- **Speculative Generality**
- **Oddball Solution**
- **Temporary Field (temporary global variable)**

Code Smells Between Classes (between .c files)

- ~~**Alternative Classes with Different Interfaces**~~
- **Primitive Obsession** #775
- ~~**Data Class**~~
- **Data Clumps**
- ~~**Refused Bequest**~~
- **Inappropriate Intimacy**
- ~~**Indecent Exposure**~~
- **Feature Envy**
- **Lazy Class**
- **Message Chains**
- ~~**Middle Man**~~
- **Divergent Change**
- **Shotgun Surgery**
- ~~**Parallel Inheritance Hierarchies**~~
- **Incomplete Library Class**
- **Solution Sprawl**

C Refactoring Tools

Coccinelle

http://coccinelle.lip6.fr/

See #690 for an example of what it can do.