Style

Note that the following are guidelines and the most important aspect of style is consistency. Strive to keep your style consistent with the project on which you are working.

Recommended Reading

The following contain good information, some of which is repeated below, some of which is contradicted below.

- http://doc.cat-v.org/bell_labs/pikestyle
- https://www.kernel.org/doc/Documentation/CodingStyle
- http://man.openbsd.org/style

File Layout

- Comment with LICENSE and possibly short explanation of file/tool
- Headers
- Macros
- Types
- Function declarations
    - Include variable names
    - For short files these can be left out
    - Group/order in logical manner
- Global variables
- Function definitions in same order as declarations
- `main`

C Features

- Use C99 without extensions (ISO/IEC 9899:1999)
    - When using gcc compile with -std=c99 -pedantic
- Use POSIX.1-2008
    - When using gcc define `_POSIX_C_SOURCE 200809L`
    - Alternatively define `_XOPEN_SOURCE 700`
- Do not mix declarations and code
- Do not use for loop initial declarations
- Use `/* */` for comments, not `//`
- Variadic macros are acceptable, but remember
    - `__VA_ARGS__` not a named parameter
    - Arg list cannot be empty

Blocks

- All variable declarations at top of block
- `{` on same line preceded by single space (except functions)
- `}` on own line unless continuing statement (`if else`, `do while`, ...)
- Use block for single statements iff
    - Inner statement needs a block

```
for (;;) {
    if (foo) {
        bar;
        baz;
    }
}
```

    - Another branch of same statement needs a block

```
if (foo) {
    bar;
} else {
    baz;
    qux;
}
```

Leading Whitespace

- Use tabs for indentation
- Use spaces for alignment
    - This means no tabs except beginning of line
    - Everything will line up independent of tab size
    - Use spaces not tabs for multiline macros as the indentation level is 0, where the `#define` began

Functions

- Return type and modifiers on own line
- Function name and argument list on next line
- Opening `{` on own line (function definitions are a special case of blocks as they cannot be nested)
- Functions not used outside translation unit should be declared and defined `static`

Variables

- Global variables not used outside translation unit should be declared `static`
- In declaration of pointers the `*` is adjacent to variable name, not type

Keywords

- Use a space after `if`, `for`, `while`, `switch` (they are not function calls)
- Do not use a space after the opening `(` and before the closing `)`
- Always use `()` with `sizeof`
- Do not use a space with `sizeof()` (it does act like a function call)

Switch

- Do not indent cases another level
- Comment cases that FALLTHROUGH

Headers

- Place system/libc headers first in alphabetical order
  - If headers must be included in a specific order comment to explain
- Place local headers after an empty line
- When writing and using local headers
  - Do not use `#ifndef` guards
  - Instead ensure they are included where and when they are needed
  - Read https://talks.golang.org/2012/splash.article#TOC_5.
  - Read http://plan9.bell-labs.com/sys/doc/comp.html

User Defined Types

- Do not use `type_t` naming (it is reserved for POSIX and less readable)
- Typedef structs
- Do not typedef builtin types
- Capitalize the type name

- Typedef the type name, if possible without first naming the struct

```
typedef struct {
    double x, y, z;
} Point;
```

Line Length

- Keep lines to reasonable length (current debate as to reasonable)
- If your lines are too long your code is likely too complex

Tests and Boolean Values

- Do not test against `NULL` explicitly
- Do not test against `0` explicitly
- Do not use `bool` types (stick to integer types)

- Assign at declaration when possible

```
Type *p = malloc(sizeof(*p));
if (!p)
    hcf();
```

- Otherwise use compound assignment and tests unless the line grows too long

```
if (!(p = malloc(sizeof(*p))))
    hcf();
```

Handling Errors

- When functions `return -1` for error test against `0` not `-1`

```
if (func() < 0)
    hcf();
```

- Use `goto` to unwind and cleanup when necessary instead of multiple nested levels
- `return` or `exit` early on failures instead of multiple nested levels
- Unreachable code should have a NOTREACHED comment
- Think long and hard on whether or not you should cleanup on fatal errors

Enums vs #define

- Use enums for values that are grouped semantically and #define otherwise.

```
#define MAXSZ  4096
#define MAGIC1 0xdeadbeef

enum {
    DIRECTION_X,
    DIRECTION_Y,
```

```
    DIRECTION_Z
};
```