



A. House Style

A.1 Correctness

These style rules ensure your code is as-correct-as-can-be with the aid of the compiler and other tools:

FLAGS Having no warnings (or errors!) when compiling and executing with the flags:

For array bounds checking, NULL pointers being dereferenced etc:

```
-Wall -Wextra -Wfloat-equal -Wvla -pedantic -std=c99  
-fsanitize=undefined -fsanitize=address -g3
```

For memory leaks:

```
-Wall -Wextra -Wfloat-equal -Wvla -pedantic -std=c99  
-g3
```

then run:

```
valgrind --leak-check=full ./myexec
```

For ‘final’ production-ready code:

```
-Wall -Wextra -Wfloat-equal -Wvla -pedantic -std=c99  
-O3
```

You can use more flags than this, obviously, but these will make sure a few of the essential warnings that commonly indicate the presence of bugs and leaks are checked. These guidelines are meant to be independent of the particular compiler used though. Sometimes it is helpful to use many compilers too, e.g. gcc and clang.

If you have unused variables (for example) in your code, it doesn’t matter whether your compiler happened to tell you about it or not - it’s still wrong !

BRACE Always brace all functions, fors, whiles, if/else etc. Somewhat controversial, this ensures that ‘extra’ lines tagged onto loops are dealt with correctly. For instance:

```
while(i < 10)  
    printf("%i\n", i);  
    i++;
```

looks like it should print out `i` 10 times, but instead runs infinitely. The programmer probably meant:

```
while(i < 10){
    printf("%i\n", i);
    i++;
}
```

GOTO You do not use any of the keywords `continue`, `goto` or `break`. The one exception is inside `switch`, where `break` is allowed because it is essential ! These keywords usually lead to tangled and complex ‘spaghetti’ coding style. I often recommend that you rewrite the offending code using functions, which **can** have multiple `return`s in them.

NAMES Meaningful identifiers. Make sure that functions names and variables having meaningful, but succinct, names.

REPC Repetitive code. If you’ve cut-and-paste large chunks of code, and made minor changes to it, you’ve done it wrong. Make it a function, and pass parameters that make the changes required.

```
int inbounds1(int i){
    if(i >=0 && i < MAX){
        return 1;
    }
    else{
        return 0;
    }
}

int inbounds2(int i){
    if(i >=0 && i < LEN){
        return 1;
    }
    else{
        return 0;
    }
}
```

might make more sense as:

```
int inbounds2(int i, int mx){
    if(i >=0 && i < mx){
        return 1;
    }
    else{
        return 0;
    }
}
```

GLOB No global variables. Global variables are declared ‘above’ `main()`, are in scope of all functions, and can be altered **anywhere** in the code. This makes it rather unclear **which** functions should be reading or writing them. You can make a case for saying that occasionally they could be useful (or better) than the alternatives, but for now, they are banned !

RETV Any functions that returns a value, should have it used:

```
scanf("%i", &i);
```

is incorrect. It returns a value that is ignored. Instead do:

```
if(scanf("%i", &i) != 1{
    /* PANIC */
```

The only exceptions are `printf` and `putchar` which do return values but which are typically ignored.

MATCH For every `fopen` there should be a matching `fclose`. For every `malloc` there should be a `free`. This helps avoid memory leaks, when your program or functions are later used in a larger project.

STDERR When exiting your program in an error state, make sure that you `fprintf` the error on `stderr` and not `stdout`. Use `exit`, e.g.

```
if(argc != 2){
    fprintf(stderr, "Usage : %s <filename>\n", argv[0]);
    exit(EXIT_FAILURE);
}
```

A.2 Prettifying

These rules are about making your code easier to read and having a consistent style in a form that others are expecting to see.

LLEN Line length. Many people use terminal and editors that are of a fixed-width. Having excessively long lines may cause the viewer to scroll to off the screen. Keep lines short, perhaps < 60 characters. However, in a similar way to the **FLEN** rule below, it's really about the complexity of the line that's the issue, not its absolute length. A programmer would generally find:

```
bool arrcleanse(cell oldarr[HEIGHT][WIDTH], cell newarr[HEIGHT][WIDTH], int h, int w)
```

a great deal easier to read than:

```
if(a < b && j++ >= szpar(e ? true : false) || h==4){
```

despite it being twice as long.

TABS Don't use tabs to indent your code. Every editor views these differently, so you have no guarantee that I'm seeing the same layout as you do. Use spaces. This also prevents issues when cutting-and-pasting from one source to another.

INDENT Indentation: choose a style for indentation and keep to it. I happen to use 3 spaces, put opening braces for functions on a new line, but at the end of `if`, `else`, `for`, `while` etc, then close them on a new line, underneath the 'i' of the `if`:

```
int smallest(int a, int b)
{
    if(a < b){
        return a;
    }
    else{
        return b;
    }
}
```

You can use any style you like, as long as it's consistent.

MAIN The code should have function prototypes/definitions first, then `main()`, followed by the function implementation. This means the reader always know where to find `main()`, since it will be near the top of the file.

CAPS Constants are `#defined`, and use all CAPITALS. For instance:

```
#define WEEKS 52
#define MAX(a,b) (a < b ? b:a)
```

FLEN Short functions. All functions are short. It's quite difficult to put a maximum number of lines on this, but use 20 as a starting point. Exceptions include a function that simply prints a list of instructions. There would be no benefit in splitting it into smaller functions. Short functions are easier to plan, write and test.

I find it more useful to think about how hard the function is to understand, rather than its length. Therefore, a 30 line, simple function is fine, but an extremely complex and dense 15 line function might need to be split up, or more self-documentation added.

A.3 Readability

Your code should be self-documenting. Comments will be written when there is something complex to explain, and only read when something has gone catastrophically wrong. In many cases clever use of coding will avoid the need for them. The compiler never sees them, so cannot check them. If you change your code, but not your comments, they can be highly misleading.

As Kevlin Henney said :

A common fallacy is to assume authors of incomprehensible code will somehow be able to express themselves lucidly and clearly in comments.

MAGIC No magic numbers. There should be no inexplicable numbers in your code, such as:

```
if(i < 36){
```

It's probably unclear to the reader where the 36 has come from, or what it means, even if it is obvious to the programmer at the time of writing the code. Instead, `#define` them with a meaningful name. Array overruns are often cured by being consistent with `#defines`.

BRIEF Comments are brief, and non-trivial. Worthless commenting often looks something like:

```
// Set the variable i to zero
int i = 0;
```

The programmer extracts no additional information from it. However, for more difficult edge cases, a comment might be useful.

```
// Have we reached the end of the list ?
if(t1->h == NULL){
```

To prevent lines from becoming too long, it is good practice to put comments above the line it refers to, not at the end of the same line.

TYPE You should use typedefs, enums and structs to increase readability.

INFIN No loops should be infinite. I'll never ask you to write a program that is meant to run forever. Therefore statements such as

```
while(1){
```

or

```
for(;;){
```

are to be avoided.

2DINDEX 2D Arrays in C are indexed `[row][col]`. Sometimes it may still work correctly, especially if you've consistently confused the two. Therefore, if you write code that indexes it `[col][row]`, or `[x][y]` it will confuse anyone else trying to understand (or reuse) your code. If you were to sketch a graph using (i, j) you'd almost certainly make i the horizontal axis, and j the vertical. Therefore, for any two variables it makes more sense to write `[b][a]` or `[j][i]`.