1. C Programming House Style

1.1 Correctness

These style rules ensure your code is as-correct-as-can-be with the aid of the compiler.

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

```
-Wall -Wextra -Wfloat-equal -pedantic -ansi -O2
```

You can use more flags than this, obviously, but these will make sure a few of the essentials warnings that commonly indicate the presence of bugs are checked.

BRACE Always brace all functions, `for`s, `while`s, `if/else` etc. Somewhat contraversial, this ensures that 'extra' lines tagged onto loops are dealt with correctly. For instance:

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

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

```
while(i < 10){
   printf("%d\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.

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

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

```
        if(scanf("%d", &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);
}
```

1.2 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.

INDENT Indentation. Be consistent. Choose a style for indentation and keep to it. I use 3 spaces rather than tabs, 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;
   }
}
```

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 `#define`d, 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 20 line function might need to be split up, or more self-documentation added.

1.3 Readability

Your code should be self-documenting. Comments will be used when there is something complex to explain, but in many cases clever use of coding will avoid the need for them.

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 `#define`s.

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){
```

TYPE You should use `typedef`s, `enum`s and `struct`s 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.

Last updated 18th October 2016.