

How (Not) To Write Comments

Andrew Clausen

June 10, 2005

First year computer science students spend much of their time doing battle with their computers, compilers and innocent text editors. Why can't the machines *just understand*? After a narrow victory, they begin second year, where convincing the computer is not enough anymore. Now they must meet the exacting standards of their tutors, who demand elegance in the form of small functions, descriptive variable names and most difficult of all: comments. This article critiques several commenting styles that new programmers often adopt. I call these *the mystic translations*, *autobiography*, *military*, *comic strip* and *haiku*. The article finishes with a recommended style, *minimalist*.

The most common style of commenting, *the mystic translations* often looks like this:

```
#define BUF_SIZE 100          /* The size of input buffers. */

/* This function uses a buffer and fgets() to read in input, and uses the
 * C library function atoi() to convert the buffer into an integer.
 */
int readint()
{
    char    buf[BUF_SIZE];          /* The buffer for input. */
    /* Reads in an input string of length BUF_SIZE into buf. */
    if (!fgets(buf, BUF_SIZE, stdin))
    {
        /* Tells the user that more input was expected. */
        printf("Unexpected end of input.\n");
        /* Exits the program */
        exit(EXIT_FAILURE);
    }
    /* Uses the atoi() library function to convert the buffer into
     * an integer.
     */
    return atoi(buf);
}
```

These students are proud of mastering the obscure machine codes of C, and fear that other programmers might be struggling with `printf()`. Fear not, these students are happy to explain every line of their code to you. Little do they realise that the intermingling of code with comments is harder to understand than each on their own. This style is also frequently used in a futile attempt to explain the workings of genuinely unintelligible code. This is always a waste of time: the only solution is to fix the poorly written code by splitting it up into smaller functions and choosing appropriate variable names.

Another popular style, the *autobiography* gives a detailed account of the student's experience with their program:

```
#define BUF_SIZE 100
```

```

/* When I first attempted to write this function I wanted to use scanf() to
 * do both the input and the conversion to an integer.  But, I couldn't
 * figure out how to get scanf() to detect if the user didn't type in
 * anything, so I used gets().  But the gets() manual page told me that
 * it is dangerous to use gets(), and I should use fgets() instead.
 */
int readint()
{
    char    buf[BUF_SIZE];

    if (!fgets(buf, BUF_SIZE, stdin))
    {
        printf("Unexpected end of input.\n");

        /* I didn't know you could exit a program from outside of
         * main().  I only just found out that you can use exit()
         * instead!
         */
        exit(EXIT_FAILURE);
    }

    /* I implemented this myself, and then found that the C library
     * has a function to do this automatically!
     */
    return atoi(buf);
}

```

Unfortunately, these comments do not help other people reading them to understand the program, nor explain what the code is trying to achieve.

The *military* commenting style rigidly follows the author's favourite standard:

```

#define BUF_SIZE 100          /* USED BY:
                             *      - readint, line 54
                             *      - readstring, line 237
                             *      - readcmd, line 421
                             */

/* FUNCTION:    readint()
 * AUTHOR:      aclausen
 * MODIFIED:    30-Mar-2005, 21:23 hours
 * PARAMETERS:  none
 * RETURNS:     int
 * USED BY:     readcmd, line 433
 */
int readint()
{
    char    buf[BUF_SIZE];
    if (!fgets(buf, BUF_SIZE, stdin))
    {
        printf("Unexpected end of input.\n");
        exit(EXIT_FAILURE);
    }
}

```

```

    return atoi(buf);
}

```

Some of the information is already available in human-friendly form: the function name, list of parameters and the return type. The authors and modification times of functions are often important for large projects, but this information is better stored in source code repositories such as the Concurrent Version System (CVS). The list of locations where functions are called can be automatically generated and navigated with tools like ctags and cscope. Meanwhile, the disciplined format of *military* comments often (but need not) neglect to explain the most important thing: what is the goal of the function? The military style can be useful for automatically generating API (Application Program Interface) documentation with tools like JavaDoc and Doxygen.

I find the *comic strip* style most entertaining. It adapts action sequences to source code:

```

/*****
 *          | DEFINES |
 *          | START   |          (C) Clausen Profit Systems, Inc.
 *          | HERE!!! |
 *****/
#define BUF_SIZE 100          /*----- maximum BUFFER SIZE!!!! ----*/

/**** READINT *****/
/* Reads in an integer! *****/
/*****/
int readint()
{
/***** TEMPORARY VARIABLES */
    char    buf[BUF_SIZE];
/***** BEGIN FUNCTION CODE :) */
    if (!fgets(buf, BUF_SIZE, stdin))
    {
        printf("Unexpected end of input.\n");
        exit(EXIT_FAILURE);
    }
    return atoi(buf);
/***** END FUNCTION CODE :( */
}

```

It's almost like Street Fighter, where Round Two starts... **NOW**! Code written in this style attempts to communicate structure that should be expressed in other ways, such as separating unrelated pieces of code into separate functions or separate files. While the theatre can be entertaining, a more minimalist approach would focus the audience's attention on the things that matter.

Haiku is another entertaining yet surprisingly popular style of comment:

```

#define BUF_SIZE 100          /* String size supremum */

/* Input int in */
int readint()
{
    char    buf[BUF_SIZE];
    if (!fgets(buf, BUF_SIZE, stdin))
    {
        /* Screen-print error */
        printf("Unexpected end of input.\n");
    }
}

```

```

        exit(EXIT_FAILURE);
    }
    return atoi(buf);
}

```

Unlike authors of *the mystic translations*, these students view source code as poetry. Apparently source code isn't complicated enough on its own, so they place extra puzzles inside the comments. Only Zen masters are worthy of reading their code, so tutors are happy to accept their place in the world and skip to the next project in the pile.

So, what comment style should you follow? I recommend the *minimalist* commenting style:

```

#define BUF_SIZE 100

/* Reads in an integer from standard input, and returns it. */
int readint()
{
    char    buf[BUF_SIZE];
    if (!fgets(buf, BUF_SIZE, stdin))
    {
        printf("Unexpected end of input.\n");
        exit(EXIT_FAILURE);
    }
    return atoi(buf);
}

```

The minimalist commenting style follows these two guidelines:

- **Explain the objective of each function.** Before each function, explain what the function does. Pretend that the audience is a programmer who is trying to decide whether or not the function solves their problem. They won't care if you used `scanf()`, `fgets()` or `read()`. They just want to know what the objective of your function is.

There are several reasons why you should focus on this information. Firstly, it is usually the first thing someone reading your code will want to know. Function names are often too terse to explain the function's purpose, and only give a rough idea. In the running example, it is unclear if from the name `readint()` if it reads from a file or standard input. Finally, it requires a little bit of thought to figure out what a function does by reading its source code. This information should be available at a glance.

Other less important information you might want to add before a function include:

- Detailed requirements on the parameters of functions. (Eg: are NULL pointers allowed?)
- Performance characteristics.
- Clever tricks or non-obvious algorithms that are relevant to the implementation of the entire function.

This guideline is sometimes summarized as “*what*, not *how*”.

- **Avoid comments inside functions.** C is an excellent language for describing how algorithms work. If your function is small function and has appropriate variable names, it should be obvious how your code achieves its objectives - if you stated them (see the previous guideline). Explaining messy code is a waste of time.

One exception to this guideline: if a clever algorithm exploits a mathematical fact, or uses some non-obvious trick that is not relevant to the design of the function as a whole, it is best to place a comment next to the relevant code.