Recommendations on writing self-documented source code

J.C.González
Max-Planck-Institut für Physik, Munich
<gonzalez@mppmu.mpg.de>

5/3/99

Abstract

In this document we explain some tricks to get fast good documentation out of the source code we write. Any piece of code should be extensively documented, both inside the code itself and outside, with manuals, reports, user guides, etc. Here we will talk about the first part of the documentation process, the self-documented source code.

Contents

1 Introduction

The documentation of every software project, big or small, is one of the most important and difficult processes in the development of that project. One of the most important parts of the documentation is documenting the code itself. The final user will not see it, probably. But the programmers who have to maintain the code, upgrade it, modify it or even find bugs (even the original developers) will have a hard life without these comments. In fact, this is what actually all the programmers are doing most of the time: maintaining old code. Quoting to S. Oualline:

> "The amount of time spent on maintenance is skyrocketing. From 1980 to 1990 the average number of lines in a typical application went

from 23 000 to 1.2 million. The average system age has gone from 4.75 to 9.4 years."

Without a good documentation, the job of the programmers will take longer and the releases will not come at time. The whole project will be delayed. And this, of course, is bad.

2 Self-documented code

The best thing one can do when writing a piece of code is write it clear, without fancy things. Most of the people only like to see obfuscated code as a kind of art, but not when they have to modify a program written by another people. Let's make some fancy numbers. One has to think that the the time a program is going to live goes as the square of the time that the programmer thought it was going to live (the program, not herself). Another approximate law could be that the number of people p who are going to *have a look* into your code is:

$$p \propto n_{\text{files}}\, t^2 \exp(-N_{\text{comm}}^2) \tag{1}$$

and, of course, the number of problems you will have, afterwards, will go like a big power of p. You realized that these were simple estimates.

In fact, although this seems to be a joke, it is true that a self-explanatory source code will make life easier for a lot of people, including the original developer. Writing with a good style is one of the most important (and uncommon) signatures of a good programmer.

Talking about style, there are a lot of different forms of making comments. You can even have a sort of hierarchy of comments: some of them used as function headers, some others to mark different sections of code, etc. They can be multi-line comments, in plain form or boxed, or they can be single-line comments. There are hundred of different forms of writing comments. Use your own. But use it.

How can we make a source code easy to read and maintain? The secret is: at least 80% of a source file should be comments. But, in addition, you can follow some recommendations to make a self-explaining code. We make some tips for that. Since almost all the code we will write will be done in C/C++, I will stick on these languages in the examples.

Main comment of your program At the beginning of your program it is recommended to put a relatively large comment, where you can see the name of the file, the author, notes about the code and, what is more important, the purpose of that piece of code. Having such a header is a convention, but usually not all the people use such thing, or they use it only some times. For example, you can easily see how some people write really nice comments at the beginning of their source code files (`.c`), with more or less details about the purpose of the code, but they don't write a single comment inside the header files (`.h`). In the Example 1 you can see how such header could look like. (Actually you don't get much information from such header — but the rest of the file is extensively commented).

Example 1. Example of comment at the beginning of a source file.

```
//////////////////////////////////////////////////////////////
//
// camera
//
// @file      camera.cxx
// @title     Simulation of the camera phase
// @desc      Program for the simulation of the camera
// @author    J C Gonzalez
// @email     gonzalez@mppmu.mpg.de
// @notes     These notes are really small, they should be
//            much longer
//
// @maintitle
//-----------------------------------------------------------
// $RCSfile: camera.cxx,v $
// $Revision: 1.11 $
// $Author: gonzalez $
// $Date: 1999/01/14 17:32:39 $
//////////////////////////////////////////////////////////////
```

When defining variables

1. Use variable names which can give you an idea of what they mean. This means, use descriptive names for them. But you can always use names like `i`, `n`, `k` as simple counters in loops (assumed you document that specific use). For example, say `number_of_photons` instead of `nph`, or `get_parameters_fit` instead of `getpfit`.

2. Always use lowercase for the names of the variables. Use them also for names of functions. Reserve the capitalized names for `enum` definitions, constants (`const`), complex types of variables (structures, classes, unions, ...) or even global variables (although you should not use them anyway). Reserve the all-uppercase names for preprocessor constants or flags (`defines`). For example, we can define a class like `Mirror_square`

   ```
   class Mirror_square {
   ...
   };
   ```

   and use it to define our array of mirrors as

   ```
   #ifdef DEFINE_NUMBER_OF_MIRRORS
   const int iNumber_of_Mirrors = 936;
   #endif // DEFINE_NUMBER_OF_MIRRORS
   ...
   ```

```
{
  ...
  Mirror_square mirrors[ iNumber_of_Mirrors ];
  ...
};
```

3. Use eventually an underscore (_) to separate words in a multiple words variable name. But, please, do not mix upper and lower case. You could say, for example, `number_of_mirrors`, but not `NumberOfMirrors` or, even worse, `numberofmirrors`. Nevertheless, don't use a very long name for a variable, like `number_of_elements_this_array_should_have`.

4. Use a one char prefix for every variable, giving an idea of what is the type of that variable (for constant values, you can either use this prefix or not). In the Table 1 you can find some of the most common prefixes used. For example, in the previous example in point 2, we defined the variable `iNumber_of_Mirrors`, with a prefix `i` of "integer".

5. Use also a suffix in the form `_units`, expressing the units in which that variable is measured (if any). For example, the surface of a circle would be defined as `float fCircle_surface_cm2`, using a prefix `f` for float, and provided our surface is in square centimeters (cm^2)

6. Document all the variables that you define in your code. Use for that purpose at least one line of comment before the definition itself. The *before* is important: I have seen people who used to put the comments *after* the definition of the variable, and since the common way is to put them before, one thinks that the comments belong actually to the *next* variable. Crazy.

Although the points 4. and 5. could be matter of discussion (for example, if you always use floats, or you specify clearly at the beginning exactly the units of all your variables, in a relatively big comment), the first two and the last one should be taken as real recommendations.

All this can be summarized in the piece of code shown in the Example 2.

Example 2. Example of variable definition using prefix as type indicator and suffix as units indicator.

```
...
// use optimized algorithms if debugging is off
#ifndef __DEBUG__
#  define __OPTIMIZED__
#endif // __DEBUG__

// define number of mirrors in the tessellated frame
const int iNum_Mirrors = 20;

// define a coordinate type (for loops)
enum Coordinate {X, Y, Z};
```

```
int get_mirror_parameters()
{
  // position of the center of the telescope
  float ftelpos_cm[iNum_Mirrors][3];

  // area of the telescope;
  float ftelarea_cm2[3];

  // maximum energy of the showers
  float fenergy_gev;

  // counter on the coordinates
  Coordinate k;

  // simple counters
  int i;

  //++ START

  // loop on mirrors
  for (i = 0; i < iNum_Mirrors; ++i ) {

    // inner loop on coordinates
    for (k = X; k < Z; ++k ) {

      // do a silly thing
      ftelpos_cm[i][k] = getvalue( i, k );

    }

  }

  ...

  return(0);
}
...
```

Table 1. Different prefixes commonly used when naming variables

c	(unsigned / signed) char
s	strings (arrays of chars)
i	int , short
u	unsigned int
f	float
d	double
p	pointer (may be with an additional letter explaining the type of value where it's pointing to.

When writing functions

1. Use also the notation explained above for the definition of variables (i.e. lowercase+underscore, with perhaps additional prefixes and suffixes for the type of the returning value). For example, you could define a function like `int iget_number_photons`.

2. Write a small header before the definition of the function, explaining what it does, what is the meaning of each parameter (and the possible range, if any), and what is the possible returning values. Normal fields to show in this header are the name of the program, the name of the module (in the case of a multi-module program), the purpose of that code, the author, and a sort of semi-detailed explanation of the global behaviour of the code. It is useful also to add a sort of history of changes. Most of the systems now use some version control system, and one of the features of this kind of programs is that they include automatically the information that the user has added when a new version of the code was released.

```
////////////////////////////////////////////////////////////
// @name get_select_energy
//
// @desc return energy range allowed for particles
//
// @var *le  Lower limit in the allowed energy range
// @var *ue  Upper limit in the allowed energy range
// @return   TRUE: we select the energy range; FALSE: we don't
//
// @author   J C Gonzalez
// @date     Wed Nov 25 13:21:00 MET 1998
// @notes    This is an example of enhanced documentation
//           (see below)
// @function
// @code
////////////////////////////////////////////////////////////
// get_select_energy
//
// return energy range allowed for showers from .phe file
////////////////////////////////////////////////////////////
```

```
int
get_select_energy(float *le, float *ue)
{
  ...
}
```

3. Use ANSI prototypes, i.e. instead of using the old form

   ```
   float power(a, b)
   float a;
   int b;
   {
     ...
   }
   ```

   use better the form:

   ```
   float power(float a, int b)
   {
     ...
   }
   ```

   If you have not seen this before, and you don't know non-ANSI C, there's no need to learn it: simply forget what I said about non-ANSI and write in ANSI C.

Consistency There is another rule, which is not easy to follow: no matter the style you use to comment your code, you should be consistent. If you decide not to use a units suffix, don't change afterwards in the same code. It will be all more homogeneous and easier to maintain.

3 Commenting the code

When writing a piece of code, most of the (bad) programmers just start typing until they have something that already could work. I don't want to comment on programming styles. But with this strategy there is something which is always missing: the documentation inside the code itself. Normally, what comes after this fast-typing is the refinement of the code. This means still no documentation at all. The code will grow and grow, and, you know, how many lines of comments do we have? Not a single one. Well, may be from time to time, the innermost part of our brain remind us that we should put some comments here and there, but this doesn't happen very often. Then you have a conference or a meeting somewhere, and when you come back you don't remember what were you doing.

There are three rules to write well commented code: first, write comments almost before every line of code; second, write more comments; and third, write comments between those comments. (Don't forget also to use from time to time a line of real code).

We could say that there are two kinds of comments: short and long comments. Normally we only use the short comments, and reserve the long comments for the *real* documentation.

3.1 Comments inside the code

Inside the code itself is where most of the comments will be placed. Follow these simple rules, and it will be easier also for you:

- Divide your code into mayor sections. For example, may be first you parse the command line parameters of your program, then read some data files, then compute a little bit and at the end present the bulk of the results. Well, these are mayor sections of your code, and you should separate them with a clearly visible comment, explaining what is the status at that moment, and what is going to do the next section of the code. A boxed multiline comment is ideal for these cases.

- Use a single comment, if possible, for each line of code. Sometimes this is simply too much, but then do not use a single comment for more than a dozen of lines or so. In this latter case, explain clearly what your code is doing.

- In the long comments, you can also emphasize things, and you should do it:

```
/*************************************************
 *        * * *  N O T E  * * *
 *
 *  Even when you write a comment you can
 *  emphasize words like *this* one.
 *  You can also simulate a kind of _underline_
 *
 *  REMEMBER, write a lot of comments.
 *
 *************************************************/
```

- As a matter of style, I use to put the single-line comments in a separate line above the line of code. Some people put it at the right of the code (I do it also sometimes). But you probably will change that line, and will be longer, and the comment will go too much to the right. Is a common rule to write lines with no more than 80 characters/line. This is due to historical reasons, but still a lot of people think that this is a good rule. Therefore, I try not only to follow the 80-characters rule, but also to write my single-line comments in a different line

3.2 Short comments

We call *short comments* to single-line comments, or multi-line comments with only few lines, explaining if few words what is doing the code at that moment (or what is going to do). In C both short and long comments have to be written between `/*` and `*/`. You can do this also in C++, but most of the cases the preferable way is using the one-line comment format: `// comment ...`. This is also the shortest way.

3.3 Long comments

We call *long comments* to one or more paragraphs of comments, enclosed between in the C form `/* ... */`. For long comments, this is the preferred way because you write paragraphs in more or less good English, and you may want to format them, and using the form `// ...` would be simply a mess.

One tip: there is no limit in the modern compilers to the number of lines you can put in a comment. There is also no limit in the number of comments inside the code. If this is so, why we simply don't write long documentation inside the code?

3.4 Bibliography

On every technical or scientific paper you will find at the end a list of bibliographic entries, in which you can get more information related to the point in which a given particular entry was cited, or related to the whole subject of the paper. Well, why shouldn't we use also such thing in our source code? We normally get algorithms, methods, ideas or even data from external source. If somebody afterwards have to maintain our code, it would be simply easiear for her/him if we write down the source of every external information. Therefore, I recommend to use a sort of bibliography at the end of the source code file. They could look like this:

```
...
// [tasch:mat]
//     I. L. Bronstein et al., "Taschenbuch der Mathematik",
//     Verlag Harri Deutsch, 1997
...
```

You will give the entry numbers (or labels) yourself, but otherwise the order is really non-important. Then, in the code itself, you can cite this entry

```
...
// Here we calculate the volume of a torus
// We use the formulae from [tasch:mat], p. 141
//   V = 2 pi2 R r2

fvolume_torus_cm3 = 2.0 * SQR(M_PI)
                    * fradius_big_cm * SQR(fradius_small_cm);
...
```

4 Enhanced documentation

There are a lot of programs, simple and complicated ones, that parse the code inside a file, extract information contained explicit or implicitly in the comments, and create well formatted documentation in the form of LaTeX or HTML documents. Most of these programs are simple scripts which help the user to maintain their own code. This is the case of SuS. We will talk about this small utility in another document.

The operational scheme of these tools is very simple: you can embed commands for the documentation script in your comments, since they will not affect

to the compilation of the program. In this way, you can have both the documentation of your code together with the code, in the same file. The documentation tool you use will interpret these commands inside your comments in order to produce a well structured and eventually pretty-printed documentation.

It is not only convenient to use such tools. It's also very useful and very easy. It is useful because most of the time, when you are writing your code, you must think on the comment you have to put there. It has to be good enough, since will be printed afterwards in a nice way (like a LaTeX report) or browsed through any HTML navigator. Therefore you are somehow forced to explain with words what you are coding. And most of the times you will even understand things better. Finally, it is also very easy to use these tools, since only with a couple of commands you can start writing a good self-documented code.

But remember, these tools will not work for you. It is you who has to write the documentation. Please do it carefully. And do it extensively as well.

References

[1] J. C. González. *SuS, a tool for source code documentation.* Max-Planck-Institut für Physik, München, March 1998. In preparation.

[2] D. E. Knuth. Mathematical writing. Report based on a course of the same name given at Stanford University, Autumn 1987.

[3] S. Oualline. *Practical C++ Programming.* O'Really & Associates, Inc., 1997.

[4] R. Stallman. GNU Coding Standards. Technical report, Free Software Foundation, Inc., 1998.