The Abuse of Comments

Michael Rutman

For some reason, everyone has heard that you can't overcomment. Nobody says why you can't, nor does anyone really say what benefits you get from large comments. They just say they are good. Well, I'll say they can be bad, and I'll show 3 types of bad comments I have come across: over commenting; to cover up bad variable names; and poorly worded comments. Following the examples of bad comments, I'll show some simple coding techniques, which most experienced programmers already know.

Most beginning CS students are taught to comment like this:

```
/****************************************
*   Written by Michael Rutman          *
*   6/26/94                            *
*   main(int argc, char **argv)        *
*   The main entry point to the        *
*   program.  Called by the system     *
*   argc is the number of parameters   *
*   argv is the array of parameters    *
*   only called at startup             *
*   Don't put anything before line 3   *
*   due to a bug in the compiler       *
****************************************/
```

This comment would get an A in any CS class across the country, even though the bulk of the comment is irrelevant to any serious programmer. This is the type of comment that gets ignored, and, as in this example, that can be very dangerous. There is very useful information buried in the comment. If you missed it, read the last two lines of the comment. This useful bit of information becomes lost in the dribble, especially when every routine has a huge pointless comment.

Some might argue that important comments should be highlited so programmers can find them easier. I maintain that all comments should be important, and having two levels of comments, important and dribble, show the danger of over-commenting.

The second type of bad comment is using a comment instead of good variable names. Consider, when code is maintained over several years and several programmers. Many times, I have seen comments along the following:

```
//count is set to the number of elements
for(count=0; array[count]; count++)
       {some code}

[more code]
return count;
```

Later, someone needs to insert some code before the return. Not wanting to clutter up the name space, the programmer decides to re-use count, thus giving:

```
//count is set to the number of elements
for(count=0; array[count]; count++)
       {some code}

[more code]

//Kludge for the first 10 elements
for(count=0; count < 10; count++)
       if (array[count] == myEnd)
              return count;

[more code]
return count;
```

While a code fragment this small makes the bug obvious, imagine a larger fragment with more logic going on. The bug is if myEnd is not in the first 10 elements, count is set to 10, not the number of elements. This bug is easily detected, and easily fixed. Here is how it is fixed.

```
//count is set to the number of elements
for(count=0; array[count]; count++)
       {some code}
numberOfElements = count;
[more code]

//Kludge for the first 10 elements
for(count=0; count < 10; count++)
       if (array[count] == myEnd)
              return count;

[more code]
return numberOfElements ;
```

Going through the code and always using numberOfElements instead of count, there is no more bug. But the comment is wrong now. Count is not always going to be equal to the

numberOfElements. Later, modifying the code yet again, a programmer will see the comment, and assume that count is always set to numberOfElements.

Instead of using a proper variable name in the beginning, the programmer relied on a comment to explain his obfuscation of his code. Well named variables and functions will usually explain more than a comment, and are less likely to become obsoleted by changes in the code. The following fragment has no comment, yet it is clear what is happening.

```
for(numberOfElements =0;
   array[numberOfElements];
   numberOfElements ++)
  {some code}
[more code]

//Kludge for the first 10 elements
for(count=0; count < 10; count++)
       if (array[count] == myEnd)
              return count;

[more code]
return numberOfElements;
```

The last abuse of comments I'd like to talk about is not giving enough information in a comment. Consider the following fragment

```
//Called in this way to avoid the bug in
libFunction

int *myVariable = (int *)-1;
libFunction(&myVariable );
```

Looks good, nice placed comment, tells future programmers something very important. However, what aspect is "calling this way" referring to? Is it setting myVariable to -1 or something else? And what will happen if libFunction is ever fixed? In a year, when another programmer has to debug this fragment, the comment will tell them something that flat out is wrong. This could lead to future bugs, as the programmer is going to assume there is a bug in libFunction.

This is really a case of a bad comment, not an unecessary one. This comment should be dated, and the description of the bug should exist. The words "A bug" means nothing in a year or two, or even to other programmers on your team. A comment is not the place to jot down a note to help remind you of something. If you are going to use a comment to explain the code, then explain the code. Never use a comment to help jog your memory. The following comment is much more useful:

```
//Due to byte alignment, libFunction looks
//at the address + 1 instead of address
//to pass in a null, we actually have to
//pass in a pointer to -1
//Michael Rutman 6/1/96
int *myVariable = (int *)-1;
libFunction(&myVariable );
```

So, if comments should not be a crutch to explain how the program works, what should one do to make code readable?

A very good way of making code easier to read is small routines. A rule of thumb is no routine should be more than 1 page long, and large screens don't count. If you cannot see the entire routine on one page, then you can't follow the logic at a glance. Everytime a programmer has to scroll the page, they must think about using the computer instead of thinking about how the code is supposed to work.

Another tool is complete variable names. Many times I have seen programmers use abbreviations. While abbreviating will save a few keystrokes, it can make the code very difficult to read. Does the variable svrs refer to servers or software version?

Here are two code segments: The first has errors, the second doesn't. How many errors can you quickly find?

```
int prcssPrgss1(int *prgss, char *prcss)
{
  int i;
  int l;
  int r = 0;

  l = strlen(prcss);
  for(i=1; i<l; i++)
    {
      (*prgss)++;
      r ^= Hash(prcss[i]);
    }
  return r;
}

int prcssPrgss2(int *prgss, char *prcss)
{
  int i;
```

```
  int l;
  int r = 0;

  l = strlen(prcss);
  for(i=l; 1<l; i++)
    {
      (*prcss)++;
      r ^= Hash(prcss[l]);
    }
  return r;
}
```

Here are the same two code segments with well named variables. Are the errors easier to spot?

```
int processProgressNoError(int *progress, char
*process)
{
  int count;
  int length;
  int result = 0;

  length = strlen(process);
  for(count=1; count<length; count++)
    {
      (*progress)++;
      result ^= Hash(process[count]);
    }
  return result;
}

int processProgressErrors(int *progress, char
*process)
{
  int count;
  int length;
  int result = 0;

  length = strlen(process);
  for(count=length; 1<length; count++)
    {
      (*process)++;
      result ^= Hash(process[length]);
    }
  return result;
}
```

In the first two code fragments, some of the errors are invisible because l and 1 look very similar. Others are just hard to spot, such as using prcss instead of prgrss. In the second two fragments, using length instead of 1 is very obvious, as is using 1 instead of count. Using length instead of count is a little harder to spot, but still obvious. Other pairs of characters with the same problems are 0/O, and I/l/1.

Using full names makes the code much easier to read. It is never a good idea to use single letter variable names. If the variable i stands for something, spell it out.

Using process instead of progress is still an easy one to miss, but nowhere near as easy to miss as prcss instead of prgrss. One solution is to make sure you don't have variable names similar to each other. This is not always possible, but in this case, these are local variables, and we can rename process to toBeProcessed. toBeProcessed is more descriptive, and much more difficult to confuse with progress.

An even better tool than good variable names are good routine names. A poor variable can make one routine difficult to follow, a poor routine name can make the flow of the entire program difficult to follow. And abbreviations in routine names can spread confusion throughout an entire program. Find the bug in the following C++ example:

```
class System
  {
    public:
      int svrs();
      ...
  };

class Connections
  {
    public:
      System  &system();
      int  svrs();
      Server  &srvr(int count);
  };

class Server
  {
    public:
      void prcss();
      int prgss();
  };

int prcssCon(Connection& conn)
{
  if (conn.system().svrs() == 1)
  for (int i = 0; i < conn.system().svrs(); i++)
    conn.srvr(i).prgss();
}
```

Now, we look at the same code with good method names. This code will not compile, and the error should be very obvious.

```
class System
  {
    public:
      int systemVersion();
      ...
  };
```

4 The Abuse of Comments

```
class Connections
  {
    public:
      System   &system();
      int  servers();
      Server   &getServer(int count);
  };

class Server
  {
    public:
      void proceess();
      int progress();
  };

int prcssCon(Connection& connection)
{
  if (connection.system().servers() == 1)
  for (int count = 0; count  <
connection.system().servers(); count ++)
    connection.srvr(count ).progress();
}
```

If you still don't see the error, it's an extra system() in the for loop. The conditional should be `connection.servers()`, not `connection.system().servers()`.

In summary, there are three common abuses of comments found in beginning programmers. The first is the over-commenting. The second is using comments instead of good variable, function, object, and method names. The last abuse is putting in a comment that only you could understand. Comments should be only used when absolutely necessary, and should always give a full and complete description.

Using short routines, good variable names, and good routine names will make your code easy to read.