University of California, San Diego
Department of Computer Science and Engineering
Scott B. Baden

CSE 100
Data Structures
Winter 1996

Programming Assignment Documentation and Style

Note: these notes are based in part on a writeup by Professor Walter A Burkhard.

While correctness is an important part of programming, a good programming style is equally important since it can lead to readable, maintainable, and modifiable programs. Correct software that is difficult to understand will usually not stand the test of time.

Writing great programs is similar to writing great essays. There are a few relatively fixed rules to be obeyed (like grammar and spelling) and a few general guidelines (like how to break paragraphs), and it takes a lot of practice to get good at it. There are essays which are clearly great, some which are clearly bad, and lots in between; those are the subjective judgment calls. Inexperienced programmers, like inexperienced writers, do best by following established principles until they develop a workable style of their own.

In this course, your programming assignments will be graded on both correctness *and* style. To give you an idea of what we will be looking for when we grade your programs, you should carefully read the following sections. If you have any questions, be sure and ask me or the teaching assistant.

Documentation

There are several reasons to document your programs carefully. First and perhaps most obvious, your partner, your grader (and at work) your employer and co-workers must be able to understand what your program does and how it works. Second, you may have to modify the program later, and without documentation your program will not make sense months or years after you have written it, without an effort not unlike that required of writing it initially. As a student in this course, it is to your advantage to make your program as clear as possible. We can't give partial credit, for example, if we can't figure out what you were trying to do.

Documentation for programs comes in two varieties: External Documentation and Internal Documentation. External documentation consists of a *user manual* and a *logic manual.* The user manual contains a concise description of the use of the program. This description discusses the nature of input data as well as the format and location of output data. The user manual for a program module contains a description of the input parameters for the methods as well as the returned values. The logic manual contains a description of your approach to solving the problem and gives the modular (if applicable) structure of your program.

Internal documentation consists of comments interspersed within the source program files. Every file must begin with a header comment that contains the name of the file, the author of the file, the date it was created, and a brief statement of contents of the file. Every function needs to be preceded by its own comment describing what it does, what parameters it is passed, what it returns, and any special notes such as side effects. The local variables each are accompanied with a descriptive comment as well. Finally some complicated pieces of code may require a few additional comments to explain the next few lines.

Style

Good style can make your programs easier to write, debug, read, and modify. Your program should be decomposed into program modules each with a unifying theme. Here are several principles of good style which you should follow whenever possible.

Use Encapsulation Appropriately

The C++ `class` construct can facilitate the maintainability and readability of your code through data hiding. But, don't overdo it either! A `struct` (with data members only) may in some cases be a simpler solution than a class; you may even choose to make a `struct` a private member of a `class`. The use of `friend` functions can lead to simpler code as well.

Define `private` member functions where appropriate. Never specify a `public` or `protected` data member, since this violates the basic principles of encapsulation. Use accessor member functions in lieu of making data members public or protected.

Keep separate declaration (`.h`) and implementation (`.C`) files for each class. Most member functions should go into an implementation file, and you may need more than one such file depending on the size of your program. The few exceptions are class constructors and desctructors and perhaps a few other one-liners, but... if the header file gets too large, start moving code into a `.C` file. Keep private member function implementations separate from public member functions.

It's a good idea to define your own copy constructors, since default copy constructors can lead to difficulties. If you have an class with embedded pointers, be sure to determine which of a *deep* or *shallow* copy constructor is appropriate. You will probably want a deep copy constructor, which copies the data that the member pointers point at. A shallow copy constructor simply copies the pointers and requires some care (e.g. reference counting) so that the destructor doesn't `delete` storage still in use. If you are deleting an array, remember the square brackets `[ ]` after the `delete` keyword.

Spaghetti Code

Try to keep control flow as simple as possible. Don't use `goto` statements unless they improve simplicity (rarely if ever). Avoid multiple returns from functions other than in exceptional circumstances. For example, if a function cannot allocate a data object, this could be an exception. Your code should be straight-line with function calls.

Reuse Code Whenever Possible

If several parts of your program need to calculate the average of five numbers, you should write a function which does that, and call it several times. Don't keep repeating the same code. But take care that in developing reusable code, you don't make it more complicated! This brings us to the next style point.

One Purpose per function

If you find a function doing several things, it probably should be broken into separate functions. Conceptually you should be able to describe the purpose of the function in a simple sentence. As a rule of thumb, functions should rarely take up more than a screenful of text, but use your own judgment. Avoid the temptation to write a function that performs several different tasks since this could lead to a multiplicity of return points. While this may lead to co re-use, doing so can compromise the readability of your code.

Avoid Cute Constructions

It is often possible to write enigmatic code which saves a line or two by using obscure aspects of the programming language. We will not think that you are a superior programmer if you do this; in fact, if it makes the program harder to read we may take off style points. There is a fine line between efficient programming and cuteness; if you think you are near that line you should point out what you are doing in a special comment.

Magic Numbers

Anytime you need to use a specific numeric value in your program, give it a name and define it as a constant at the beginning of the function, program, or file. That way people reading the program will know what the number means, and if its value changes, you won't have to go through and look for every place you used it. For example if an array size is 512, don't put a phrase "`if (i < 512)`" in your program. Instead define "`ArraySize`" to be a constant, and use the name in the code "`if ( i < ArraySize )`" instead. You can be assured that *all* array sizes will change during the lifetime of the code. Use a distinct manifest constant for each array unless for very compelling logical reasons the two arrays must have the same length.

Using magic numbers makes software extremely difficult to maintain. Do not use them. Make sure your constant names describe their application; for example

```
const unsigned TwentyTwo = 22 ;
```

is not particularly useful.

Meaningful Identifier Names

A variable representing an array of grades, for example, should be called something like "`grades`"; it will be much easier to recall what the variable stands for. If a variable stands for an "arbitrary" number such as a loop counter, then names like "`i`" are fine. You don't have to needlessly create "meaningful or cute" names when there is nothing there.

Correctness

There are degrees of correctness, a program may operate correctly some inputs and not others. A program which works no matter what the user enters is "more correct" than one which crashes if an illegal value is used. The following list progresses from least to most correct:

1. Program Does Not Compile.
2. Program Compiles, But Does Not Run.
3. Program Runs On Some Data. This might mean that for some values, the program produces incorrect responses, or that it crashes.
4. Program Runs For Normal Cases. For example, your program correctly reproduces answers for the test cases supplied, but fails for abnormal values. .
5. Program Handles All Cases Robustly. No matter what data are entered, the program produces either the correct response or a suitable error message. This might be viewed as industrial grade software.

Testing is related to correctness. You should provide test input data files and results for successful runs of your programs.

It is good practice to use familiar names for the files in your homework directory. For example,

```
README
makefile
homework.c
```

Be sure you have only one makefile present in the directory—the old game of random selection comes into play!

What To Hand In

All the files for a given assignment should be in subdirectories of your home directory, named `HW1`, `HW2`, `HW3`, `HW4`, etc. as appropriate. We will depend upon you to submit all the required files for the proper operation of your make file. However, you need not submit a copy of the standard library files as well as files which we provide. Each exercise will must be accompanied with a `README` file and a `makefile`. `README` must contain the author's name(s), login name(s) and the name of the make file as well as any last minute instructions to the graders. The user and logic manuals must be present. We will use your make file to create an executable version of your program. It is essential that all files, except for the standard library headers and object modules, accessed within your make file be present in your homework directory.

If your program does not compile or execute properly, *you must tell us in writing what doesn't work.* We will not give partial credit to incorrect programs unless you do this. You may put this sort of description in the `README` file. Here are a few examples of this variety of note.

> I can't get the program for exercise 1.7 to compile correctly. Whenever I compile it I get an error message saying that I have an undeclared variable.
>
> The program crashes if I enter numbers larger than 5 for the array size. I tried to track this down by checking the space allocation, but I can't find anything wrong.