

Introduction

These notes provide guidelines for internal program documentation and style. Although they are intended for student programmers, style skills will carry over into professional life after school. In fact, most professional programmers would consider these standards to be less than minimum requirements for commercial-quality software documentation. While geared mainly for a block structured language such as Pascal or C, most of the points discussed are applicable to any programming language. We will not address the "how" of program specification, design, or testing; these topics are the domain of the Computer Science curriculum.

The essence of good programming style is communication. Good style in programming is roughly as difficult to learn as good style in English. In both cases, the document has no value if it does not convey its meaning to the reader. Any program that will be used must be maintained by somebody - and that somebody must be able to understand the code by reading it. Any program that needs debugging will be easier to debug if the creator carefully explains what's going on.

Within the program text, programmers have three primary tools for communicating their intentions: comments (explanation for the program); clear variable names, constants, expressions and subroutines (the words of the program itself); and white space and alignment (the organization of the words in the program). Each of these aspects aid communication between the program writer and the program reader (which includes the program writer at debug time - so you as a program writer have a stake in good style too!).

I would advise anybody who expects to spend a lot of time at a keyboard to become a proficient typist, and learn the touch-typing method. If you never had a typing class in high school, there are many books available that will let you teach yourself.

Students who are poor typists may feel that adding comments to their program will increase the time spent when writing a program. Those students will quickly find that typing is the easiest part of programming. Finding and fixing syntax errors also becomes quite easy after a little practice, since the compiler gives you at least a clue as to how and where you went wrong. The time consuming part for many programmers is debugging. When you debug a program, you switch from being a program writer to being a program reader. This is where good style and commenting can save you many hours of hardship. The rules of programming style outlined here were developed by programmers who learned the hard way that using good style is a self-defense measure that minimizes bugs.

General Guidelines

The most useful things to know about program documentation are "what" and "when". In general, you should include comments explaining what every function does, what every variable does, and an explanation for every tricky expression or section of code.

But, there is much to good style beyond comments, as we shall see. "When" is easy - comment it before you write it.

Whenever you declare a constant or variable, include a comment.

Use named constants rather than "magic numbers". This will promote readability and make modifications easier. Remember: the same literal constant may have several different meanings throughout a program.

Avoid the use of global variables altogether. They are a bane when debugging and do not improve performance significantly.

Whenever you start to write a function, first write a comment explaining it. That will help to make clear in your mind what you are about to do. If you find the explanation for a function difficult to write, it's a sure sign that you have done a poor job of structuring the program.

The same is true for function parameter lists. Write comments explaining the significance and/or use of each parameter. Again, finding this to be difficult is a sure sign you do not yet have a good understanding of what the function is to do, and how it is to do it.

Make good use of the language's support for user-defined types (such as `struct` types in C/C++). Custom-designed data types can help the organization of a design and improve the readability of an implementation.

Avoid obscure programming language constructs and reliance on language-specific precedence rules. It is often better to force precedence by use of parentheses since this leaves no doubt as to meaning.

In general, if you had to look up some rule or definition, your readers most likely will too. Whenever you do write a difficult expression, or some other tricky business, if you found it difficult to write you should expect that it will be difficult to understand. So, add a comment.

Use ISO Standard C++ exclusively. Use the new standard header files in preference to the old pre-standard headers. Never mix old and new style headers in the same program. Use the `string` type in preference to C-style `char` arrays.

The most important rule of style is: be consistent! If you adopt some method for variable naming or indenting, stick to it throughout your program.

The Program Header

All programs should include, at or near the beginning of the program, a comment block with at least the following information: the programmer's name and appropriate identification, the date of writing (and possibly dates and purpose of major modifications), the name of the program, the computer and operating system for which it was written. Include information about compiling the program (i.e., the compile line commands). Also include a "usage" line showing how the program is called and the various command line arguments. The bulk of the program header will often be a description of the program. This serves as a roadmap that others can follow in order to understand the program. At a minimum, include an explanation of how the program works, and the major data structures and algorithms used. If you have more than one module, they should each contain a program header block describing the workings of that module. If you know about any bugs in your program, it is a good idea to include a comment in the program header explaining what they are. You might also want to keep a running "TODO" list in the program header comment. If you keep a header block "template" available, you can easily copy this template into your program when you begin. An example program header block in Pascal might look as follows:

```
// PROGRAM NAME - threshold
//
// PROGRAMMER - A. Hacker, 001-00-2001
//
// USAGE:
// threshold <image-file> <num-rows> <num-cols> <thresh-value>
//     where <image-file> contains an image array of size
//     <num-rows> X <num-cols>, and <thresh-value> is the
//     desired pixel threshold value.
//
// SYSTEM - Visual C++ 6.0, Service Pack 4.0 on NT 4.0
//
// DATE - Started 08/01/1997
//     Phase I completed 11/14/1997
//
// BUGS - Program crashes if the number of bytes in
//     <image-file> is less than <num-rows> X <num-cols>.
//
// DESCRIPTION - This program counts the number of pixels in an
//     image (represented by an array) whose value
//     is greater than a specified threshold.
//
```

Function Documentation

The functions of a program (or module) define the structure of that program. Good program structure can be viewed as a business hierarchy: the president (`main`) delegates tasks to the next level of workers, who in turn delegate subtasks to their underlings. Each function should have one clearly defined task. Writing the comment describing that task before writing the function can help to keep you straight here. Functions should be short enough so that their full meaning can be grasped as a unit by the reader.

Normally, one page of code represents the limit of intellectual complexity that a reader can grasp.

Every function should be preceded by a comment describing the purpose and use of that function. Similar to the program header, each function should have a comment block with standard sections such as function name, parameters, description, and calling functions.

If you decide to modify a function during program development, be sure to modify the comments as well to keep them accurate.

Below is an example of a function comment header block. Note that this example indicates in the header the purpose of each parameter to the function, and what is returned.

```
//////////////////////////////////// CountBigPix
// Purpose: counts and returns the number of pixels
//           in image (of size rows X cols) with value greater
//           than thresh.
// Parameters:
//   Image:   array of Pixel var. containing the image data
//   Rows, Cols: the number of rows and columns for image
//   Thresh:  the Pixel threshold value - count pixels
//           greater than this
//
// Returns:   number of elements of Image that exceed Thresh
// Called by: main()
// Calls:    none
//
int CountBigPix(const Pixel Image[], int Rows, Cols,
                Pixel Thresh) {
    . . .
}
```

One of the greatest sources of bugs is in the interface between functions.

It is a common mistake to have type mismatches between the formal parameters and the actual parameters. This may be insidious if the mismatch is masked by a default type conversion.

It is also common to misorder parameters in the call. You should always be scrupulous in checking for such problems when writing your program, and you should always check these early on when debugging. Note that misordering may or may not result in a compile-time error.

One way to minimize the problem is to keep the number of parameters to any one function at a minimum. Seven or more parameters for a function is probably too many - having this many parameters is a good indication that you have structured your program in the wrong way, or that you need to reconsider the flow of information within your program.

One way to reduce the number of parameters is to group logically connected data into a single parameter through the use of records and structures. For example, if you have a program that uses coordinate points, it probably makes more sense to group the coordinates and values into a single `struct`, instead of passing two separate pieces of information to a function.

It is also a good idea to adopt a standard ordering, such as all parameters supplying information to the function appearing before any reference parameters returning information from the function.

Pass parameters by reference only when the function must modify the actual parameter. There are only two ways that a function can screw up a variable in another function: global variables and variables passed by reference. The problem in both cases is that you increase the chance that something will be changed in a way that you didn't expect. By definition (since you didn't expect it), this is hard to debug. If your function does change a formal parameter, you should document this in the function header comment.

If pass by reference is not appropriate, pass parameters by value or by constant reference. Large parameters (arrays, `struct` variables, `string` variables) should usually be passed by constant reference rather than by value to avoid the time and memory overhead of copying.

Indentation and White Space

You should notice that in the code examples I take care to skip spaces or line things up in a special way. Adding "white space" in this way makes your program much more readable. At least two blank lines should separate your functions. It is good to put additional blank lines at appropriate places within your source code to separate major sections, such as to separate declarations from the beginning of the statements. You can also use blank lines to separate logical sections of the code.

A new level of indentation should be used at every level of statement nesting in your program. The amount of indentation is up to the programmer - there are many different styles. Commonly, programmers will indent a fixed number of spaces at each level. More spaces make the indentation clearer, but they also limit the amount of information on a line after several levels of indentation. The minimum number of spaces at each indentation should be 3, and many programmers use a tab mark (typically 8 spaces).

Where should the opening and closing braces go? Some programmers put the opening brace on the same line as the control statement, others put it on the next line. It is generally good practice to align a closing brace either with the matching opening brace, or with the beginning of the control statement that owns the block.

Here are some examples.

```
if (x < 0)
    x = 0;                // 3 spaces fixed indentation style
else {
    while (x > 0) {      // opening brace on same line as while
        P(x);
        x = x - 1;
    }                    // end while (x > 0)
}                        // end else
if (x < 0)
    x = 0;                // indentation matches keyword length style
else
{
    while (x > 0)
    {
        P(x);
        x = x - 1;
    }                    // end while (x > 0)
    y = y + 1;
}                        // end else
```

Nested if-then-else statements are often treated differently, to avoid too much indentation.:

```
if <expression1> {
    <statements>
}
else if <expression2> {
    <statements>
}
else {
    <statements>
}
```

Examine source formatting in your textbook, or good language references for ideas.

Often there are "pretty-printing" utilities available to programmers that automatically indent programs. For C programmers under UNIX, both the utility `cb` and the EMACS text editor provide automatic indenting.

You should not put more than one statement on the same line. Also, it is good practice to keep lines to less than 80 characters. Lines longer than this should be broken into multiple lines, and indented so as to line things up in a logical way, such as:

```
average = (image[x-1][y-1] + image[x][y-1] + image[x+1][y-1] +
           image[x-1][y]   + image[x][y]   + image[x+1][y] +
           image[x-1][y+1] + image[x][y+1] + image[x+1][y+1]) / 9;
```

Identifiers

Pick identifiers (i.e., names for variables, functions, records, etc.) that communicate the named object's purpose to the reader. You also want to pick identifiers in such a way that a mistake when typing will not likely result in another identifier used within your program. In this way, a typo will be caught as an error by the compiler (undeclared variable) instead of resulting in a difficult to find "logic" error.

In general, single-character identifiers are bad since they violate both of these rules. The variable names `O` and `P` are not usually very informative, and a mistyped `O` can easily become a `P`. However, in two cases it is traditional to use one letter variable names. In certain mathematical expressions it is appropriate to use traditional variable names such as `X` or `Y`. It is also appropriate to use `I`, `J` and `K` for loop variables and array indices - but for no other purpose!

Since all the reserved names in C/C++ (and most common languages) are strictly lower-case, it is best if your own identifiers are not.

Many programming languages (such as Pascal and C) allow for arbitrarily long identifiers. Be careful here, since some compilers will only use the first six or eight characters to distinguish between names.

There are a number of rules for good use of variables. An easy one to follow is: never change the value of a loop variable within that loop. In other words, don't do the following:

```
int i;
for (i = 1; i <= 10; i++) {
    i = i + 1;
}
```

This is an instance of a general principle for variable use: don't overload the logical meaning of a variable. By using `I` as the loop variable, the programmer is signaling to the reader something about `I`. By assigning to the variable, a second meaning is introduced. The program is easier to understand if a second variable is assigned to and used.

Use variables at the smallest possible level of scope. One implication of this rule is that you should minimize the use of global variables.

Some designers argue there is a place in good programming style for global variables, and that is for a body of knowledge that will be acted on by many sections of the program, and which is in some sense the major essence of that program. That opinion is widely discounted in modern software engineering.

In any case, don't use global variables as a convenient means to communicate between two functions.

Be consistent about the use of local variables with the same name throughout the program. If "I" is used as a loop variable in one function, don't use it as something else in another function.

Depending on the programming language, the compiler may or may not distinguish upper and lower case letters. However, a reader certainly can distinguish the difference, and mixing cases correctly can give the reader additional clues about a name. You should adopt a standard usage for case.

One suggestion is to have all constants entirely in upper case, all created types (typedefs, records, structures) with only the first letter upper case, and all variables in mixed case, capitalizing the first letters of subwords (`CurrentIndex`).

It is also informative if you use common suffixes and prefixes for related variables, such as `inFile`, `outFile`, `errorFile`. Beware that 0 and 1 look a lot like O and l.

Every identifier in your program should have a comment describing its use. Typically, this comment appears at the end of the line containing the declaration. You should avoid declaring more than one variable on the same line unless all variables on that line have very similar use. Here is an example of variable declarations in C/C++:

```
const int MAXSIZE = 1000000; // maximum image size
Pixel Image[MAXSIZE];      // The image buffer
int Rows, Cols;            // Number of rows and columns in image
int I, J;                  // Current row and column
int Thresh;                // Threshold: count bigger pixels
int BigPixCount;           // Total number of pixels in image
```

Constants and Enumerated Types

When you type a mathematical expression such as

$$fVal = 7 * x + 2$$

the numbers 7 and 2 are truly numbers. However, most programmers only rarely use true numbers. More likely, they are dealing with numbers that are logical quantities, such as the size of a disk block (1024) or Monday (day 1 in the week).

Of great importance (but often neglected by weaker programmers) is the use of constants and enumerated types, both of which are supported by C/C++ and most other languages in some form.

Constants allow the programmer to give a name to a number. In this way, the meaning of the number can be captured in the constant name, thus providing more information to the reader. Another advantage is that it is easy to change the value of the constant - you only need to change it at the definition.

Enumerated types allow the programmer to declare a variable with a limited range of values. These values can be logical names denoting internal state information. For example, if you want a variable to store a day of the week, instead of storing an integer value of 1 for Monday, you can store the more meaningful value `Monday`.

Program status variables (i.e., state variables) and case statement indices should never be integer values. They should always have logical names, defined either through constants or enumerated types. You should carefully consider every number that you type in your code, and substitute a name for it unless you have a good reason not to do so.

The more that you limit the possible values for a variable (through the use of enumerated types), the less chance for error. Enumerated types also give error checking compilers a chance to work in your favor, as well as supplying more information to the reader.

Types: Structures and typedef Statements

The most difficult part in writing many programs is defining the various record types (structures and classes in C++). This is because the record definitions define the data structures for the program. You should give special care to naming both the record and its subfields, even more care than to naming single variables. Below is a C/C++ example using records and enumerated types. Even without comments, the intention should be pretty clear. Note that I gave the record field the "good" name at the expense of the name for the range and enumerated types, since the field name will be used a lot within the program code.

```
enum WeekdayType {Mon, Tue, Wed, Thur, Fri, Sat, Sun};
enum MonthType   {Jan, Feb, Mar, Apr, May, Jun,
                  Jul, Aug, Sep, Oct, Nov, Dec};
typedef int DayType; //valid range 1 to 31
struct Date {
    MonthType   Month;
    DayType     Day;
    WeekdayType Weekday;
    int         Year;
};
```

Block Statements

Block statements (the group of statements between braces in C/C++) deserve a comment. This comment should indicate what is happening in the block, such as the action for each pass through the loop, or what action is being executed in the `else` part of an `if` statement. In addition, at the end of the block, a comment should indicate what block is being closed. Here is an example in C/C++.

```
while (listPtr != NULL) {
    /* sum all values on the list */
    Total += listPtr->Value;
    listPtr = listPtr->Next;
} /* while (listPtr != NULL) */
```

`goto` Statements

There are few good uses for a `goto` statement. It is not uncommon for the class instructor to ban `goto` statements altogether. The traditional legitimate use for a `goto` is to allow the programmer to escape from deep nesting when a special case (usually an error) has been encountered.

For C/C++ programmers, using a `goto` for "escape" is never justified since you can return from any point in a function, and the keywords `break` and `continue` allow you to break out of a loop, and `exit()` allows you to terminate the program at any point. Advanced C++ programmers may use exceptions to achieve cleaner handling of such a situation.

FORTRAN programs may use `goto`'s to simulate while statements, or C-style `break` and `continue` statements.

If you should ever feel the need to use a `goto` statement, you should comment the reason thoroughly in the program! And you are probably making a poor decision.

Final Remarks

Despite all that was written above, remember that terseness is a virtue. A smaller body of words is easier to comprehend. When documenting a program, say only what needs to be said without leaving important things out. For example, the following comment is not informative:

```
count++; // Increase count by 1
```

Beginners will have difficulty deciding what can safely be left out. So, beginners should err on the side of completeness.

The original draft of this document was written by Dr. Clifford Shaffer of the Department of Computer Science at Virginia Tech. Over time the document has been updated and modified by William McQuain.

This document will continue to be updated from time to time. If you have any suggestions for how it may be improved, please bring it to the attention of Mr. McQuain or Dr. Shaffer.