CODING STYLE IN C

COMMENTS

Documentation aids software maintenance and makes software easier to modify. Organizations spend *much more* on maintaining software than they do on the original software development, so good documentation saves resources, and should not be overlooked in software development.

You should put comments in these places:

· At the beginning of each file stating what functions are in this file, who wrote them and the date, then an additional line for each person who later modified the code in this file, what functionality they changed/added and the date. (This history is sometimes one of the most valuable things in the file.)

· before each function in the program (including main)

· data structure definitions

· global variable definitions

· tricky steps within a function

· before each block of statements

· anything out of the ordinary

The opening / of all comments should be indented to the same level as the code to which it applies.

If you put a comment on the same line as code, set it off from the code with a few tabs; don't continue such a comment across multiple lines.

The size of the comment should be proportional to the size of the code that it refers to.

Comments should describe *what* the code is doing, not *how* it’s being done.

TYPES AND DECLARATIONS

Explicitly define types as often as possible.

Think about the order of the fields.

Try to keep related fields grouped.

Within groups of related fields, pick some uniform scheme for organizing them, for example alphabetically or by frequency of use.

When all other considerations are equal, place larger fields first, as C's alignment rules may then permit the compiler to save space by not introducing holes in the structure layout.

NAMING CONVENTIONS

#define symbolic constants should be in ALL CAPS.

Variable and function names should be meaningful, and reasonably descriptive.

Names should be chosen to make sense when your program is read.

Variable identifiers should be noun phrases; shortening/abbreviation is acceptable, if someone reading your code would be likely to understand the meaning.

Single letters as identifiers are only acceptable in very limited circumstances (such as loop counters, for example), and only where the use of the variable will be clear without a descriptive name

Use some consistent scheme for naming related variables.

Functions should be named for what they do, not how they do it.

Function names should reflect what theyreturn.

INDENTATION AND LAYOUT

Each {} statement block set of statements, except in the outermost block of a function, should be indented. Use tabs for indentation.

Keep lines to a reasonable length – generally, no more than 80 characters.

Avoid unnecessary curly braces, that is, ones which do not improve readability or aid in determining the structure of the code.

Use spaces around keywords.

Use spaces around binary operators, except . and -> (for they are equivalent in effect to array subscripts), and the “punctuation" operator ','.

Don't use spaces around unary operators, except sizeof and casts.

Don't parenthesize things unnecessarily (that is, when the parentheses do not aid readability, or do not make the order of operations, especially in complex expressions, clear).

If an expression gets too long to fit in a line, break it before a binary operator. Put the operator at the beginning of the next line to emphasize that it is continued from the previous line. Don't add additional indenting to the continued line.

EXPRESSIONS AND STATEMENTS

Feel free to use a FOR loop where some of the parts are empty.

If a FOR statement gets too long to fit on a single line, turn it into a WHILE; if your loop control is that complicated, it probably isn't what FOR is for (pun intended).

Always put braces following IF, WHILE, and FOR statements. If you do this, you never have to remember to put them in when you want more than one statement to be executed, and you make the body of the if statement more visually clear.

IN GENERAL

Create your code so that it is modular, in understandable blocks of code.

Functions should be short and sweet; generally, a function should do one “step” in the high-level algorithm. If the whole algorithm is complex, each of the high-level steps may be divided into multiple functions. **Functions which consist of one or two lines of code are not uncommon in C! Therefore, the fact that a function requires very little code if it is written separately is NOT a justification for incorporating it into another function.**

Include sanity checks in your code… “and you thought this couldn’t happen” kind of scenarios.

Use goto sparingly. Two harmless places to use it are to break out of a multilevel loop, or to jump to common function exit code. (Often these are the same places.)

Avoid sloppiness and inconsistency. Decide what your style is and follow it precisely and consistently.

The standard library is your friend. There's no excuse for writing code which already exists there.

Fancy algorithms are buggier than simple ones, and they're much harder to implement.

Use simple algorithms as well as simple data structures.

Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming even though C is a functional language.