A Standard for C Code

Following a coding standard is part of professional programming. This enhances the readability of your code, it improves the quality and makes it easier for other programmers to read and modify your code.

Names

To make the code self-documenting, choose meaningful names for variables. Abbreviations may be used so long as they are widely accepted. A good test of names is: *can you read your code to a fellow programmer over the phone?*

For names that consist of multiple words, capitalize the first letter of each word.

Distinguish classes of names as follows:

Functions, Macros, Types, Classes: First letter uppercase (eg. GetInput(), LengthType, Compute()).

***Constants*:** All uppercase, separate words with '_' (eg. MAX_LINE_LEN, PI, VOTING_AGE)

***Variables*:** First letter lowercase (eg. roomMessDistance, inBuf, myId, windowHt, wallWidth)

Names should differ in more than one character, especially if they are of the same type. E.g., for the transmit and receiver buffers, *txBuf* and *rxBuf* differ in only the first character which occurs on adjacent keys on the keyboard. *txBuf* and *rcvBuf* is a better choice.

Use the following abbreviations to identify particular names:

Type	Defined type (e.g. typedef struct {...} MsgType;)
Ptr	Pointer (e.g. bufPtr, msgPtr, pktPtr)
Fl	Boolean (e.g. moreFl)
Str	String (e.g. promptStr)
Chr	Character (e.g. inChr, outChr)
Tab	Table (e.g. relayTab, relayTabPtr)
Num	Number (e.g. numCourses) ["No" could be confused with the negative]
Ctrl	Control (e.g. CTRL_C)
Cmd	Command (e.g. LastCmd)
Cnt	Count (e.g. wordCnt)
Que	Queue (e.g. inBufQuePtr)
Len	Length (e.g. roadLen)

Internal Documentation

Apart from external documentation such as pseudo-code, flow-charts, state transition diagrams, function-call hierarchies, and prose, the program files should contain documentation. Begin **each file** with a comment including the following fields:

```
/*****************************************************************
 * sort.c – for sorting integers              filename with one-line description
 * Purpose: uses bubble-sort algorithm...     purpose in detail
 * Compilation: use the supplied makefile     Instructions for compiling
 * Revision history:                                Chronological list of changes/bug-fixes
 *    A. Programmer, 7/7/77
 *          released version 1.0
 *    C. Debugger, 8/8/88
 *          fixed stack overflow with null input
 *    Eager B. Eaver, 9/9/99
 *          added ANewProc() to support 3-D
 * Bugs:                                            Known bugs/limitation/testing to be done
 *          The program occasionally crashes when two users
 *          access the database simultaneously during the new moon.
 *****************************************************************/
```

Declare **each variable** on a separate line, followed by an inline comment explaining the purpose of the variable. Use

```
    char *inBuf;   // buffer for received keystrokes
    char *outBuf; // buffer for text going to the printer
```

rather than

```
        char *inBuf,  *outBuf;  // input and output buffers
```

If there are a large number of variables, group them in blocks by function, and alphabetically within each block. Note: temporary variables such as loop indices need not follow some of these rules.

Preceding **each function**, include a comment block as follows:

```
/*****************************************************************
 * GetInput - get input from the keyboard.
 * Args:  Stores the string in the buffer buf, max size is bufSize
 * Returns: number of characters stored in buf
 *                    or -1 on error.
 * Method:         a brief description if necessary.
 * Bugs: list known bugs and limitations
 * To be done: if anything
 *****************************************************************/
int GetString(char *buf, int bufSize)
  {
    ...
  }                             /*  End of GetString()       */
```

Within the body of the function, on separate lines at the start of **each major block** (if, while for, switch), describe briefly the purpose and peculiarities of the block. For obscure statements, include an inline comment.

Avoid obvious comments such as:

```
    i++;              /* increment i  */
```

Layout

Indent the code according to the following scheme and use blank lines to indicate breaks in the flow of control. This improves the readability.

```
while (moreFl)                  /* The main loop, terminates when done */
  {
    if (i == 2)
            DoSomethingAppropriate();
    else
            DoSomethingElse();

    for (j = 0; j < maxFile; j++)       /* Mumbo-jumbo for each file */
        {
                        total += table[i].wordCnt;
                        i      = j + k;
                        cnt    = j – 1;
        }
  } /* while (moreFl) */
```

Useful Features

Some C language features that will enhance the quality of your code:

Header files: collect macro, type, constant and global variable declarations and prototypes for public functions in one or more .h include files. Never include code in .h files. Group logically related functions into separate .c files. Use a utility such as *make* to automate rebuilding the program.

Information hiding: declaring a function static makes it private to the module (i.e., file) in which it is declared. Likewise for data. In a header file, define *#define PRIVATE static* and use it for private functions and data:

```
PRIVATE int myCount;
PRIVATE void LocalFunc();
```

Function prototypes: use these to enable the compiler to check for consistency of arguments. In a header file, include function prototypes for all public functions. Remember to use void for functions that do not return any value.

Enumerated types: use *enum* rather than a sequence of *#defines*. This is less error-prone and enables the compiler to check type consistency.

Type casts: use explicit typecasts to avoid warning messages from the compiler about operands of different types.