A Standard for C Code

Dr. T.A. Gonsalves
Distributed Systems & Optical Networks Lab
Department of Computer Science & Engineering
I.I.T., Madras - 600 036

23 June, 1994

Preamble

The use of uniform coding and documentation standards makes it easier for one person to read and understand the code written by another. This is important in a group project, or when a program is likely to be maintained and/or enhanced by someone other than the original author(s).

Names

To make the code self-documenting, choose meaningful names for variables. Abbreviations may be used so long as they are widely accepted and do not hinder intelligibility. In view of the limitations of some linkers, ensure that global names are unique in the first 8 characters. 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, do not use '_'. Distinguish classes of names as fol-

lows:

```
Functions, Macros, Types, Classes:
   First letter uppercase
Constants:    All uppercase, separate words with '_'
Variables:    First letter lowercase
```

Some examples:

```
     #define MAX_LINE_LEN 500

     typedef int LengthType;
     typedef *Length LengthPtr;

     int GetString(char *buf, int bufSize);
     LengthType x, y, roomMessDistance[MAX_ROOMS];
     char inBuf[MAX_LINE_LEN];
     unsigned hostID;
```

Remember, a name is typically typed a few times, and read many times. If you do not like typing, use the power of a text-editor to ease your burden: type an abbreviation, then do a global search-and-replace; or use the automatic abbreviation expansion capabilities of editors such as Emacs.

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

Use the following to identify particular names (:

```
Type   Defined type (e.g. typedef struct {...} MsgType;)

Cls    A class name

Obj    An object of a class
```

```
Ptr   Pointer (e.g. bufPtr, msgPtr, pktPtr)

Addr   Address (e.g. ioBaseAddr)

Flg   Boolean (e.g. moreFlg)

Str   String (e.g. promptStr)

Chr   Character (e.g. inChr, outChr)

Tx   Transmit

Rcv   Receive

Buf   Buffer (e.g., txBuf, rcvBuf)

Msg   Message

Pkt   Packet

Tab   Table (e.g. relayTab, relayTabPtr)

Num   Number (e.g. numPkts) ["No" could be confused with the
  negative]

Ctrl   Control

Cmd   Command

Que   Queue (e.g. inBufQuePtr)

Len   Length (e.g. pktLen)

Hdr   Header (e.g. hdrLen)

Trl   Trailer

Grp   Group
```

Types

In cases such as accessing hardware, or constructing packets, the precise length of a variable is important. The default char, int and long types which may be implemented differently in different compilers. Hence, use the following (basic.h):

```
    #ifdef  MSC
typedef     char      CHAR;
typedef     unsigned char   U_CHAR;
typedef     char      BYTE;
typedef     unsigned char   U_BYTE;
typedef     int       INT16;
typedef     unsigned int    U_INT16;
typedef     long      INT32;
typedef     unsigned long   U_INT32;
    #elseif UNIX
typedef     char      CHAR;
typedef     unsigned char   U_CHAR;
typedef     char      BYTE;
typedef     unsigned char   U_BYTE;
typedef     short      INT16;
typedef     unsigned short  U_INT16;
typedef     int       INT32;
typedef     unsigned int    U_INT32;
    #endif
```

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:

```
/*****************************************************************
 *
 * file name with one-line description of contents
 * util.c - a utility for ...
 *
 * Purpose of the code in this file - in detail
 * Purpose: this utility does ... using algorithm ...
 * Compared with alternative algorithms ...
 *
 * Instructions for compiling
 * Compilation: use the supplied makefile
 *
 * Chronological list of all major changes and
 1
 * bug fixes

-----------

1. For multi-file programs, it may be better to have the revision
   history entirely in one file (such as the one containing main(), or
   a header file, or history.c.
 * Revision history:
 *    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, tests to be done, etc
 * Comments:
 * The program occasionally crashes when two users
 * access the database simultaneously during the
 * new moon.
 *
```

```
 ************************************************************/

Declare each variable (except temporaries such as loop indices) on a
separate line, followed by an inline comment explaining the purpose of
the variable.  Use

     char *rcvBuf;
     char *txBuf;

 rather than

     char *rcvBuf,
  *txBuf;

  Where appropriate, group variables in blocks by function, and
alphabetically within each group.

Preceding each function, include a comment block as follows:

/*****************************************************************
 *
 * GetString - get a string from the moon.
 * Stores the string in the buffer buf.
 * bufSize is the size of the buffer.
 *
 * Returns: number of characters stored in buf
 *  or -1 on error.
 *
 * Method: a brief description if necessary.
 *
 * Bugs: if any
 *
 * 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, 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. Each new block should be indented 4 spaces from the previous one. Open/close braces should be in the same column and both indented 2 spaces from the previous level. Label the closing brace of a large block with a comment. Use blank lines liberally to indicate breaks in the flow of control. Use lines longer than 80 columns only infrequently. Continuation lines for multi-line statements should be indented. In a sequence of assignment statements, line up the '=' signs.

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

    fprintf(stderr, "This is %u words from %s.\n",
table[i].wordCount, table[i].fileName);

/* Mumbo-jumbo for each file */
    for (j = 0; j < maxFile; j++)
      {
total += table[i].wordCount;
```

```
i      = j + 1;
count  = j - 1;
      }
  } /* while (moreFlg) */
```

Useful Features

2

Some C language features that are useful but not always used:

---

2. Some of these may not be available in non-ANSI-standard compilers.

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.

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. Remember to use void for functions that don't return any value.

Conditional compilation: use #ifdef to select machine/OS/compiler dependant code, making your code as portable as possible. Some common names:

```
 DOS       PC/MS-DOS
 UNIX       Generic Unix
 BSD       BSD Unix
 SUNOS       SUNOS, a variant of BSD
 SYSV       System V Unix
 ANSI       ANSI terminal
 IBMPC       IBM PC hardware
 MSC, QUICKC Microsoft C compilers
 GCC       GNU C compiler
 TCC       Turbo C
```

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 files and use a utility such as make to automate rebuilding the program.

Globals: for global variables, use the following scheme. Place all globals in a header file, e.g., prog.h, each preceded by the keyword PUBLIC. In the main file, e.g., prog.c, insert the following:

```
      #define PUBLIC
      #include "prog.h"
```

In all other .c files, insert the following:

```
      #define PUBLIC extern
      #include "prog.h"
```

This results in storage being allocated for the globals exactly once in prog.c and the globals being declared external in all other source files. Any changes to the global declarations need be made in only one place.

External Documentation

As you work, prepare and keep up-to-date the following for each function/module that you work on (some may not be relevant for simple functions that do not call other functions). All documents and code must contain your name and date of each significant revision.

1. Specification - inputs, purpose, side-effects, outputs

2. Implementation - details on major data structures, pseudo-code for algorithms, highlighting anything unusual, to-be-done and known bugs or limitations

3. Test plan - either exhaustive or selective with justification

4. Test log - each time the function/module is "released" for use by others, the log must contain an entry certifying that you have run the complete test plan. During testing, do not use other untested modules.

5. List of files with a one-line description of each

6. List of constants, types, variables and functions with a one-line description of each and the file in which it is defined. Indicate whether each is PUBLIC, PRIVATE (to a file) or LOCAL (to a function).

7. Function call-tree, i.e., for each function, a list of all functions that it calls (use the Microsoft calltree utility or equivalent).

8. Function called-by tree, i.e., for each function, a list of all functions that call it (use the Microsoft calltree utility or equivalent).

Version Control

Each software package is to have a version number consisting of four parts, e.g., 2.3.15c. The first three are mandatory, the fourth is optional.

Major version This should change only when significant new features are added to the software. The change must be agreed upon by all members of the development group. E.g., 2.3.15c has major version 2.

Minor version This should change when minor new features are added to the software. The change must be agreed upon by all members of the development group. E.g., 2.3.15c has minor version 3. Note that 2.3.10 is a later version than 2.3.9.

Patch level This is incremented every time a bug is fixed. This can be incremented by the individual applying the patch. E.g., 2.3.15c has patch level 15.

Environment code
The software may run under different environments such as OS, computer, etc. Alternatively, there may be different versions for different customers. Each such environment is identified by a unique letter. E.g., 2.3.15c has environment code c. Note: environment code is optional.

We recommend the use of the Revision Control System (RCS) package for maintaining different versions, especially with multi-member teams. RCS is available at no cost and runs on Unix, DOS, OS/2 and other systems.

Homily

Make it a practice to include documentation on the above lines as you write the code for the first time. Adding documentation later is much more tedious and prone to error (you may no longer be quite sure why you did something months ago) and often gets short shrift in the rush to get the last bug out before the deadline for completing the project.