C Programming Standards for CLAS Software

Lee Crawford  
Department of Physics  
Florida State University  
Tallahassee, FL 32306

D. P. Heddle  
Department of Physics  
Christopher Newport College  
Newport News, VA 23606

This document establishes the guidelines for CLAS software written in "C". Whenever possible, we strive for consistency with the FORTRAN77 software standards as defined in The Hitchhiker's Guide to the Galaxy: CLAS Software Manual, [1] to which this document can be considered as a supplement.
1 Introduction

This CLAS-Note establishes the conventions and standards adopted by the Software Working Group (SWG) for "C" code written for use by the CLAS collaboration.

Note that the restrictions on programming languages for CLAS software, as described by Joyce and Dennis [1] are still in effect. FORTRAN77 is always the language of choice for CLAS software. This document addresses the rare instances when "C" is chosen because FORTRAN77 is indisputably impractical. Permission to use "C" (or "C++") in official CLAS software must be obtained on a case-by-case basis after consulting with the SWG. In particular, "C" may not be used for purely aesthetic reasons.

To its credit, the "C" community has, over the years, adopted its own de facto standards. In some cases we deviate from these standards for purposes of conforming with our own FORTRAN77 guidelines; in other cases we deviate from them because we cannot appreciate their utility.

2 "C" Coding Standards

Coding standards are important for readability, extendibility, debugging and maintenance. Standards, if successful, are also helpful in gleaning information from a quick glance at a section of code; Information such as the scope of variables and the location of function definitions.

There is a misconception about "C" that it belongs in the class of "structured" programming languages, such as Pascal. In truth, one must work as hard in "C" to write structured programs as in FORTRAN. If anything, poorly written "C" code is more impenetrable than bad FORTRAN. In addition, the strict type checking present in Pascal is absent in "C". For these and other reasons it is important to establish guidelines.

In the subsections that follow, we present the SWG guidelines for the various components of a "C" code.

2.1 Header Files

Header files should use the standard "C" convention for preventing double inclusion. Basically, given a header file named header.h one should structure the file as:

```c
#ifdef _HEADER_H
```
# define _HEADER_H

... body of header.h...

#endif /* _HEADER_H */

Also, header files should not include other header files.

2.2 Function Names

The function naming convention is identical to that established in *The Hitchhiker’s Guide...* for FORTRAN77 code. In particular, each function name begins with a six character prefix in the following format: the first two letters represent the detector package or software package (the “noun”), the next three characters represent the analysis section or action (the “verb”), and the sixth character is an underscore. What follows the underscore (the function name “proper”) should be meaningful, and WordCaps without additional underscores is the preferred policy. Thus, an example routine to create a database file might look something like:

```c
int DbGen_CreateDatabase(char *filename, int type)
```

This convention distinguishes coded functions from intrinsics as well as X-windows and related libraries (XLib, It, Motif, ...).

Registered character sequences for the function prefixes are tabulated in *The Hitchhiker’s Guide...* and not reproduced here.

2.3 Function Prototype Conventions

The most important rules governing function prototypes are:

- All functions must return an explicitly typed returned value, which may be of type void.
- All functions must have explicitly typed arguments.
- All functions to be used outside their immediate scope must have a prototype declared in an associated header file.
- Complex datastructures should be passed by reference. This is a memory conserving practice.
• It is preferred that all functions return an error status (either as the
value returned by the function or one of the function arguments).
This rule may be ignored for exceedingly trivial functions. Functions
performing numerical calculations of any complexity or functions re-
quiring memory allocations are never exempt.

An example of an acceptable function prototype is:

ErrStat  EdGen_AddSubMenu(char *Label,
               MenuStruct *menu,
               MenuStruct *submenu)

where ErrStat is a type with obvious interpretation and MenuStruct is
a complex datastructure. One would typically use this with a predefined
global constant NoErr in the following manner:

if (EdGen_AddSubMenu(theLabel, ParentMenu, SubMenu) == NoErr)
    No error statement
else
    call error handling function

It is also acceptable to check for the occurrence of errors (rather than the
non-occurrence, as in the example above). In either case, one must never
assume a particular representation of boolean values.

Finally, we require that functions of a similar nature with subsets of
common arguments should take their arguments in a common order. An
example of this is found in X-windows programming, where the rectangu-
lar region specification parameters x, y, width, height are used in many
routines and always specified in the same order.

2.4 Constant Declarations

It is widely held that "magic" numbers in programming code can make pro-
gram maintenance an abominable chore. The old "C"-language had a means
of solving this problem by using a preprocessor capable of symbol/macro
expansion. The ANSI "C" standard [2] defines a new means of declaring
numerical constants; The user declares the value and type of his constant
and then uses it in the manner of a regular variable. Typically (in old "C")
a user might define a constant as in:

#define PI 3.141592653 /* WRONG*/

and at each point in the code where one enters "PI" that text is re-
placed by "3.141592653". The problem is that the replacement may occur
at any place in the program without ensuring that type-checking is being
performed. The solution to this problem in ANSI "C" is the const qualifier.
The right way to declare the constant PI is:

const double PI = 3.141592653; /* CORRECT*/

The upside to this usage is that the value now carries type information.
The downside is that there is, on occasion, a little more overhead incurred
in some usages. However, the pitfalls are fewer with this declaration style.
Also, we are assured that the compiler will not allow the address-of (&)
operator to be applied on any variable of type const, thus protecting against
indirect changes to the value through pointers.

2.5 Macro Declarations

We have adopted the rule that macro names should be in upper-case with
no underscores. With this unique convention, macro invocations will be
easy to recognize. In particular, macros will never be confused with intrinsic
functions (which are always lower-case) or the WIMP (Windows Icons Menus
Pointers) libraries, which use mixed case. Here are a few examples:

#define MAX(A,B) (((A) > (B)) ? (A) : (B)) /* usual max*/
#define MIN(A,B) (((A) < (B)) ? (A) : (B)) /* usual min*/
#define MALLOC(X) ((X *) malloc(sizeof(X))) /* for any type*/

2.6 Global Variables

Because of the richness in the "C" scoping rules, "C" actually has different
levels of global variables. For purposes of simplicity, we identify all variables
that are not strictly local to one function as being global.

Names for global variables adhere to the same rules as function names.
This is consistent with the CLAS FORTRAN77 guidelines.
2.7 Local Variables

We impose no rules of the naming of local variables other than the obvious policy that the names be meaningful. However, there is an implied rule regarding consistency (see the section on Style Consistency, below.) There are also rules governing the declaration of local variables (see the section on Comments, below.)

2.8 Type Declarations

We require typedefs to be WordCaps without underscores. This avoids collisions with function names and global variables which always have at least one underscore (the sixth character). In addition, we follow the sensible convention that typedef'ed pointers to typedef'ed structures merely append a "Ptr" to the typename:

/* All CLAS graphical objects are DrawObj structures stored
   in one or more doubly linked lists */

typedef struct drawobj *DrawObjPtr; /* pointer to ced object */

typedef struct drawobj
{
   Drawable drawable; /* window where it's drawn */
   short int objecttype /* specifies object type */
   ...
} DrawObj;

2.9 Comments

First and foremost we follow the "Joyce-Dennis" rule-of-thumb: the number of comments should approximately equal the number of lines of code. In addition, we have the following guidelines for comments:

- Inline, aligned comments are preferred
• In most cases, only one variable should be declared per line, and an inline comment should be included.

• Single line typedefs should have inline comments similar to variable declarations. typedefs for structures should have a summary comment above the definition explaining the utility of the structure. Each field should be defined on a separate line with an inline comment, as in the example above for the DrawnObj structure.

• Files and functions should contain statement-of-purpose/revision comment blocks, as described in the Hitchhiker’s Guide.... For example:

```c
/*
 * File: ced_dispatch.c
 *
 * Summary: Provides toplevel dispatches
 * for menus and dialog boxes. The callbacks
 * are contained in this file; most callbacks
 * are if-else blocks that call other functions
 * based on what item was selected or
 * what button was pressed.
 *
 * Contributors:
 * D. Heddle HEDDLE@CEBAFVAX (804) 249-7432
 * G. Lin GEBIG@CEBAFVAX (804) 249-7102
 *
 * Revision History:
 *
 * DATE PROGRAMMER COMMENTS
 * --------------------------
 * 1/3/91 DH Initial Version
 * 1/12/91 GL Rewrote DH's pidgin C
 */
```

• While the number of comment lines is roughly equal to the number of code statements, in many circumstances a single comment above a simple and well defined block is more helpful than a strict one-comment-per-line policy:
/
** Count the number of characters in string "string". */

c = 0;
while (*string++)
    c ++;

is preferred to:

c = 0;  /* Zero integer var c */
    /* While value at string != 0, inc string */
while (*string++)
    c++;  /* Increment c */

Any additional rules described in The Hitchhiker's Guide apply to "C" codes as well.
Exceptions for the one declaration per line rule are made for intimately related variables, e.g.:

    int x, y;  /*pixel location of upper left*/
    short width, height; /*width and height in pixels*/

is acceptable. Another exception would be for trivial local variables used for looping:

    int i, j, k;  /*do-loop indices*/

2.10 Miscellaneous Rules

- When using the if-else construction, you should use the compound statement format if EITHER the if-clause or else-clause contains multiple statements. This means that:
if (something)
    myFunc (a);
else {
    Inc (a);
    myFunc (a);
}

must be written:

if (something) {
    myFunc (a);
} else {
    Inc (a);
    myFunc (a);
}

Adherence to this policy avoids if-then-else ambiguities.

- Null bodies of for and while statements must appear alone on the following line to emphasize that the null body was intended. Thus use:

    while (something --)

    ;

instead of:

    while (something --);

- Multistatement lines may be used rarely and in special circumstances. For example, in the standard method of invoking the It macro ItSetArg:

    n = 0;
    ItSetArg(args[n],ImLabelString,ImSt); n++;
    ItSetArg(args[n],ImSet,State); n++;

- NULL-ness of pointers should be tested using the negation operator:
if (!MyPointer)
    NULL pointer statement
else
    nonzero pointer statement

- Finally, as always. the use of global variables should be minimized.

2.11 Style Consistency

In certain areas we have not established strict rules. For example, we have no guidelines for the naming of local variables, other than that they should be meaningful. Another example is that we don't require a standard way for indenting the braces of a compound statement following, say, an if conditional (which is one of the most religious questions in "C").

We do ask for consistency. If you are an anti-underscore rebel, and prefer not to use them except where required, then don't make exceptions; Declare all local variables without underscores. If you follow the Kernighan and Ritchie school of indentation:

\[
\text{if (type \neq \text{PARENS}) \{ } \\
\text{    statement} \\
\text{    statement} \\
\text{\}}
\]

then by all means be fanatical.

3 Code Maintenance

3.1 ReadMe Files

The source directories for all "C" projects should contain a ReadMe file. This file should contain comprehensive installation instructions and recent user-manual entries not appearing in the printed documentation.
3.2 make

All CLAS "C" codes must be distributed with a *makefile*. The unix *make* utility is indispensable for software development at the level of complexity likely to be found in a CLAS software project. In addition to automating code updating, a *makefile* documents all files needed to build an executable as well as the required libraries.

The SWG has not established a naming convention for makefiles.

3.3 lint

All CLAS "C" codes must be "lint-free".

3.4 Inherited Responsibility

When responsibility for maintenance of an existing code is inherited, it is our policy that the programming style of the original author (assuming that it meets the guidelines specified in this document) should be adopted for all subsequent modifications.

4 Acknowledgements

It is a pleasure to acknowledge the helpful comments of Larry Dennis, Bryon Ehlman, Don Joyce, Barry Preedom and Greg Ricardi. One of us (D.H.) would also like to acknowledge the grammar checking program that found 18 instances of the use of passive voice in the first version of this document.

References