Proposed C Coding Standards for Spring

Douglas Niehaus

April 1989

Header Files

Header files should be created for any information that is common to more than one C source file, or must be coordinated across more than one product. In most cases this information consists of structure definitions, constants, and macros. It can also include external declarations that are required to use a product.
The relevant standards are:

- Macros and Constants should be declared using all capital letters.
- Nesting of include files should be avoided.
- Constants associated with structure elements should be mentioned in the comment for the element, and declared following the structure.

The header files themselves should be structured. At the file level this means that data structures, constants, and macros associated with a particular subsystem should be grouped together in a header files. More than one header file may be required if the system is particularly large, or if a distinction needs to be made between internal and external information. Along these lines we should create files such as:

- ipc.h
- task.H
- sched.h
- memory.h

C Source Files

The C standards are designed to make all code follow the same layout rules, which lets and experienced eye flow easily over the text. Different people will work on code at different times, and several people may modify the same code if they are working together. All of these are reasons to write code in a consistent manner.

- The SCCS ID should appear at the top of the file.
- Include statements should appear below the SCCS ID.
- Global variables should appear next.

- Global variables should start with a capital letter.
- Variables should each be declared on their own line, with a comment to the right about what each is for. The exception to this are obvious scratch variables.
- Alphabetizing long variable declaration sections makes them easy to consult, but is not terribly important.
- Each routine should be defined using the following format:

```
type
function(a, b)
int      a;            /* First parameter */
int      b;            /* Second parameter */
{
}
```

  So, for example:

```
int
name(fp, tp)
FILE        *fp;      /* output file pointer */
task_t      *tp;      /* ptr to task struct to output */
{
}
```

  This permits a search for a routine definition with the command:

```
grep "^name" *.c
```

- Routine names should, generally, be of the form:

```
op_obj_descr();
```

  Where "op" is the operation being done, "obj" is the object or area this routine addresses, and "descr" is anything more that seems helpful. This is a fairly loose rule, since some situations do not fall into this pattern. Some examples that do would be:

```
pr_db_summary()
write_db()
read_db()
```

- Local declarations should give external functions first, then static variables, followed by the rest.
- Comments should generally be blocks preceding a section of code, not comments at the end of the line. This encourages a style that narrates each relevant section of the code, and is not visually cluttered. For instance:

```
/*
 * This is a block comment before
 * a section of code, or the definition
 * of a routine.
 */
```

Single line comments should appear only on the right of a line they are commenting. They probably should be limited to commenting variable definitions. An exception would be where a particular section of code is very difficult to understand, and specific lines require individual explanation.

- The **if** statement, **while** loop, and **for** loop should all use the brackets in the following way:

```
if ( boolean ) {
    statement;
}

if ( boolean ) {
    statement;
}
else {
    statement;
}

while ( boolean ) {
    statement;
}
```

Note that it is a good idea to use the brackets on all **if** statements, even those with a single statement. This is because people often add statements, forgetting to add the brackets.

Makefiles and SCCS

Makefiles are an important source of information about a product, and should be written clearly. The use of standard targets is also a great help to someone trying to learn some new code. Makefile standards have been presented in my SCCS document.

The makefile is as much a part of the code as the C source. The makefile gives the information about how a product is put together, and how to assemble the pieces of an object is just as important as the pieces themselves. As an example, consider how useful a VW would be as a pile of component parts.

Similarly, preservation in the SCCS data base of particular versions of the files, and through slists, particular versions of products as produced by particular *sets* of files is vital. Sharing code by trading path names into one another's directories is unreliable and time consuming. It is more reliable to place shared information in standard places that are not associated with a particular person. Shared source code is generally in the form of header files, shared object code in the form of object libraries. The information is even more reliable if the shared information is under SCCS control.

Libraries

As we find commonly useful subroutines, we should make them available to one another through libraries. This involves some extra work, but is an invaluable aid to other people trying to use our code. The makefile of a product can construct a library as an intermediate step, and then install it in a global area. The documentation for the library can be minimal, but it must include a description of each routine. The description should give the parameters and the routine's purpose.

These libraries should then be installed in a standard area where all other developers can reference them in their makefiles. The global areas for sharing are discussed int he next section.