C Style and Coding Standards for SunOS

Bill Shannon

Version 1.8 of 96/08/19.

ABSTRACT

This document describes a set of coding standards and recommendations that are local standards for programs written in C for the SunOS product. The purpose of these standards is to facilitate sharing of each other's code, as well as to enable construction of tools (e.g., editors, formatters) that, by incorporating knowledge of these standards, can help the programmer in the preparation of programs.

This document is based on a similar document written by L.W. Cannon, R.A. Elliott, L.W. Kirchhoff, J.H. Miller, J.M. Milner, R.W. Mitze, E.P. Schan, N.O. Whittington at Bell Labs. It also incorporates several items from a similar paper written by S. Shah. The current version was derived from an earlier version written by the author and Warren Teitelman.

C Style and Coding Standards for SunOS

Bill Shannon

Version 1.8 of 96/08/19.

1. Introduction

The scope of this document is the coding style used in writing C programs for the SunOS product. To the extent that we do adhere to a common style, it will be easier for several people to cooperate in the development of the same program. It also will facilitate understanding and maintaining code developed by someone else.

More important than the particular coding style used is *consistency* of coding style. Within a particular module, package, or project, a consistent coding style should be used throughout. This is particularly important when modifying an existing program; the modifications should be coded in the same style as the program being modified, not in the programmer's personal style, nor necessarily in the style advocated by this document.

This document discusses ANSI C only briefly, and C++ is hardly mentioned. A future version of this document will discuss ANSI C more fully, describing when to use such new features as `const`. In addition, rules for writing C code that must interact with C++ code, and vice versa, will be covered. Style rules for pure C++ programs will be left to a separate document.

Of necessity, these standards cannot cover all situations. Experience and informed judgment count for much. Inexperienced programmers who encounter unusual situations should consult 1) code written by experienced C programmers following these rules, or 2) experienced C programmers.

2. File Naming Conventions

UNIX† requires certain suffix conventions for names of files to be processed by the `cc` command[1]. Other suffixes are simply conventions that we require. The following suffixes are required:

- C source file names end in *.c*
- Assembler source file names end in *.s*
- Relocatable object file names end in *.o*
- Include header file names end in *.h*
- Archived library files end in *.a*
- Fortran source file names end in *.f*
- Pascal source file names end in *.p*
- C-shell source file names end in *.csh*
- Bourne shell source file names end in *.sh*
- Yacc source file names end in *.y*

† UNIX is a trademark of Bell Laboratories.

1. In addition to the suffix conventions given here, it is conventional to use 'Makefile' (not 'makefile') for the control file for *make* and 'README' for a summary of the contents of a directory or directory tree.

- Lex source file names end in *.l*

3. Program Organization

The choices made in organizing a program often affect its understandability and ease of maintenance. Many modern programming languages provide mechanisms for defining the modules of a program. Although C does not have these language features, they can be approximated using the techniques and conventions described below.

A module is a group of procedures and data that together implement some abstraction, for example, a symbol table. Those procedures and data that can be accessed by other modules are called *public*. Procedures and data known only inside the module are called *private*. Modules are defined using two files: an *interface* file and an *implementation* file. The interface file contains the public definitions of the module. The implementation file contains the private definitions and the code for the procedures in the module.

In C, the role of an interface file is provided by a header file. As mentioned above, the name of a header file ends in *.h*. The implementation file for a particular header file often has the same root name, but ends in *.c* instead of *.h*. Private definitions in the implementation file are declared to be `static`.

As an example, here are the interface and implementation files for a symbol table package:

symtab.h

```
/*
 * Symbol table interface.
 */

typedef struct symbol Symbol;
struct  symbol {
        char    *name
        int     value;
        symbol  chain;
};

Symbol  *insert();
Symbol  *lookup();
```

symtab.c

```
/*
 * Symbol table implementation.
 */

#include "symtab.h"

#define HASHTABLESIZE   101

static Symbol hashtable[HASHTABLESIZE];
static unsigned hash();
```

```
/*
 * insert(name) inserts "name" into the symbol table.
 * It returns a pointer to the symbol table entry.
 */
Symbol *
insert(name)
        char *name;
{
        ...
}


/*
 * lookup(name) checks to see if name is in the symbol table.
 * It returns a pointer to the symbol table entry if it
 * exists or NULL if not.
 */
Symbol *
lookup(name)
        char *name;
{
        ...
}


/*
 * hash(name) is the hash function used by insert and lookup.
 */
static unsigned
hash(name)
        char *name;
{
        ...
}
```

When the implementation of an abstraction is too large for a single file, a similar technique is used. There is an interface file, or in some cases, several interface files, which contain the public definitions that clients of the package will see, and a collection of implementation files. If it is necessary for several implementation files to share definitions or data that should not be seen by clients of the package, these definitions are placed in a private header file shared by those files. A convention for naming such header files is to append '_impl' to the root name of the public interface file, e.g., *symtab_impl.h*, although this convention isn't widely followed. Generally speaking, such implementation header files should **not** be shipped to customers[2].

4. File Organization

A file consists of various sections that should be separated by blank lines. Although there is no maximum length requirement for source files, files with more than about 3000 lines are cumbersome to deal with. Lines longer than 80 columns are not handled well by all terminals and should be avoided if possible[3].

2. Traditionally many kernel implementation header files have been shipped, for use by programs that read `/dev/kmem`. Following this tradition for kernel header files is allowed but not required. In other areas of the system, shipping implementation header files is strongly discouraged.

3. Excessively long lines which result from deep indenting are often a symptom of poorly organized code.

The suggested order of sections for a *header* file is as follows (see the Appendix for a detailed example):

1. The first thing in the file should be a comment including the copyright notice. This comment might also describe the purpose of this file.
2. The second thing in the file should be an `#ifndef` that checks whether the header file has been previously included, and if it has, ignores the rest of the file (see section 5).
3. Next should be a `#pragma ident` line[4].
4. If this header file needs to include any other header files, the `#include` statements should be next.
5. Next should be a guard to allow the header file to be used by C++ programs.
6. If it did not appear at the beginning of the file, next there should be a block comment describing the contents of the file[5]. A description of the purpose of the objects in the files (whether they be functions, external data declarations or definitions, or something else) is more useful than just a list of the object names. Keep the description short and to the point. A list of authors and modification history is **not** appropriate here.
7. Any `#define`s that apply to the file as a whole are next.
8. Any `typedef`s are next.
9. Next come the structure declarations. If a set of `#define`s applies to a particular piece of global data (such as a flags word), the `#define`s should be immediately after the data declaration.
10. Next come the global variable declarations[6]. Declarations of global variables should use the `extern` keyword. (Never declare static variables in a header file.)
11. Finally come the declarations of functions. All external functions should be declared, even those that return `int`, or do not return a value (declare them to return `void`).
12. The end of the header should close with the match to the C++ guard and the match to the multiple inclusion guard.

The suggested order of sections for a *.c* file (implementation file) is roughly the same as a header file, eliminating unneeded constructs (such as the multiple inclusion and C++ guards). Don't forget the copyright notice and `#pragma ident`.

After all the declarations come the definitions of the procedures themselves, each preceded by a block comment. They should be in some sort of meaningful order. Top-down is generally better than bottom-up[7], and a breadth-first approach (functions on a similar level of abstraction together) is preferred over depth-first (functions defined as soon as possible after their calls). Considerable judgement is called for here. If defining large numbers of essentially independent utility functions, consider alphabetical order.

5. Header Files

Header files are files that are included in other files prior to compilation by the C preprocessor. Some are defined at the system level like <*stdio.h*> which must be included by any program using the standard I/O library. Header files are also used to contain data declarations and `#define`s that are needed by more than one program[8]. Header files should be functionally organized, i.e., declarations for separate

---

4. The form `#pragma ident` is preferred over the obsolete and less portable form `#ident`.

5. This comment sometimes appears between items 3 and 4. Any of these locations is acceptable.

6. It should be noted that declaring variables in a header file is often a poor idea. Frequently it is a symptom of poor partitioning of code between files.

7. Declaring all functions before any are defined allows the implementor to use the top-down ordering without running afoul of the single pass C compilers.

8. Don't use absolute pathnames when including header files. Use the <*name*> construction for getting them from a standard

subsystems should be in separate header files. Also, if a set of declarations is likely to change when code is ported from one machine to another, those declarations should be in a separate header file.

It is often convenient to be able to nest header files, for example, to provide the user with a single header file which includes, in the right order, a number of other header files needed for a particular application. However, some objects like typedefs and initialized data definitions cannot be seen twice by the compiler in one compilation. Therefore, to provide for the possibility that two master header files will both include the same header file, each header file should contain a check for whether it has previously been included. The standard way of doing this is to define a variable whose name consists of the name of the header file, including any leading directories, but with characters that are otherwise illegal in a variable name replaced with underscores, and with a leading underscore added. The entire header file is then bracketed in an `#ifndef` statement which checks whether that variable has been defined. For example, here is a header file that would be referenced with `#include <bool.h>` which defines the enumerated type `Bool`:

bool.h

```
/*
 * Copyright (c) 1993 by Sun Microsystems, Inc.
 * All rights reserved.
 */

#ifndef _BOOL_H
#define _BOOL_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

#ifdef  __cplusplus
extern "C" {
#endif

typedef enum { FALSE = 0, TRUE = 1 } Bool;

#ifdef  __cplusplus
}
#endif

#endif /* _BOOL_H */
```

It is a requirement that exported system header files be acceptable to non-ANSI (K&R) C compilers, ANSI C compilers, and C++ compilers. ANSI C defines the new keywords `const`, `signed`, and `volatile`. These keywords should only be used within ANSI C compatible parts of the header file. The biggest difference between ANSI C and K&R C is the ability to declare the types of function parameters. To handle both forms of declarations for external functions, the following style should be used.

place, or define them relative to the current directory. The −I option of the C compiler is the best way to handle extensive private libraries of header files; it permits reorganizing the directory structure without altering source files.

```
#if defined(__STDC__)
extern  bool_t bool_from_string(char *);
extern  char *bool_to_string(bool_t);
#else
extern  bool_t bool_from_string();
extern  char *bool_to_string();
#endif
```

Some people find it helpful to include the parameter types in comments in the K&R form of the declaration.

```
extern  bool_t bool_from_string(/* char * */);
extern  char *bool_to_string(/* bool_t */);
```

While ANSI C allows you to use parameter names (as well as types) in function declarations, and many people find that the names provide useful documentation of the function parameters, the names must be chosen extremely carefully. A user's `#define` using the same name can render the function declaration syntactically invalid.

The `extern "C"` guard is required in the header file for C++ compatibility. In addition, the following C++ keywords should not be used in headers.

```
asm      friend     overload    public     throw
catch    inline     private     template   try
class    new        protected   this       virtual
delete   operator
```

Note that there are many additional rules for header files that are specified by various standards, such as ANSI C, POSIX, and XPG.

6. Indentation

Only eight-space indentation should be used, and a tab should be used rather than eight spaces. If eight-space indentation causes the code to be too wide to fit in 80 columns, it is often the case that the nesting structure is too complex and the code would be clearer if it were rewritten. The rules for how to indent particular C constructs such as `if` statements, `for` statements, `switch` statements, etc., are described below in the section on compound statements.

7. Comments in C Programs

Comments should be used to give overviews of code and provide additional information that isn't readily available in the code itself. Comments should only contain information that is germane to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it should reside should not be included as a comment in a source file. Nor should comments include a list of authors or a modification history for the file; this information belongs in the SCCS history. Discussion of nontrivial design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It's too easy for such redundant information to get out-of-date. In general, avoid including in comments information that is likely to become out-of-date.

Comments should **not** be enclosed in large boxes drawn with asterisks or other characters. Comments should never include special characters, such as form-feed and backspace.

There are three styles of comments: block, single-line, and trailing. These are discussed next.

7.1. Block Comments

The opening /* of a block comment that appears outside of any function should be in column one. There should be a * in column 2 before each line of text in the block comment, and the closing */ should be in columns 2-3 (so that the *'s line up). This enables *grep* ^.* to catch all of the block comments in a file. There is never any text on the first or last lines of the block comment. The initial text line is separated from the * by a single space, although later text lines may be further indented, as appropriate.

```
/*
 * Here is a block comment.
 * The comment text should be spaced or tabbed over
 * and the opening slash-star and closing star-slash
 * should be alone on a line.
 */
```

Block comments are used to provide English descriptions of the contents of files, the functions of procedures, and to describe data structures and algorithms. Block comments should be used at the beginning of each file and before each procedure. The comment at the beginning of the file containing `main()` should include a description of what the program does and its command line syntax. The comments at the beginning of other files should describe the contents of those files.

The block comment that precedes each procedure should document its function, input parameters, algorithm, and returned value. For example,

```
/*
 * index(c, str) returns a pointer to the first occurrence of
 * character c in string str, or NULL if c doesn't occur
 * in the string.
 */
```

In many cases, block comments inside a function are appropriate, and they should be indented to the same indentation level as the code that they describe.

Block comments should generally contain complete English sentences and should follow the English rules for punctuation and capitalization. The other types of comments described below will more often contain sentence fragments, phrases, etc.

7.2. Single-Line Comments

Short comments may appear on a single line indented over to the indentation level of the code that follows.

```
if (argc > 1) {
        /* get input file from command line */
        if (freopen(argv[1], "r", stdin) == NULL)
                error("can't open %s\n", argv[1]);
}
```

Two single-line comments can appear in a row if one line isn't enough, but this is strongly discouraged. The comment text should be separated from the opening /* and closing */ by a space[10]. The closing */'s of several adjacent single-line comments should **not** be forced to be aligned vertically. In general, a block comment should be used when a single line is insufficient.

10. Except for the special lint comments `/*ARGSUSED*/` and `/*VARARGS`*n*`*/` (which should appear alone on a line immediately preceding the function to which they apply), and `/*NOTREACHED*/`, in which the spaces are not required.

7.3. Trailing Comments

Very short comments may appear on the same line as the code they describe, but should be tabbed over far enough to separate them from the statements. If more than one short comment appears in a block of code, they should all be tabbed to the same tab setting.

```
if (a == 2)
        return (TRUE);          /* special case */
else
        return (isprime(a));    /* works only for odd a */
```

Trailing comments are most useful for documenting declarations. Avoid the assembly language style of commenting every line of executable code with a trailing comment.

Trailing comments are often also used on preprocessor `#else` and `#endif` statements if they are far away from the corresponding test. Occasionally, trailing comments will be used to match right braces with the corresponding C statement, but this style is discouraged except in cases where the corresponding statement is many pages away[11].

8. Declarations

There is considerable variation at Sun in the formatting of declarations with regard to the number of declarations per line, and whether using tabs within declarations makes them more readable.

8.1. How Many Declarations Per Line?

There is a weak consensus that one declaration per line is to be preferred, because it encourages commenting. In other words,

```
int     level;          /* indentation level */
int     size;           /* size of symbol table */
int     lines;          /* lines read from input */
```

is preferred over:

```
int     level, size, lines;
```

However, the latter style is frequently used, especially for declaration of several temporary variables of primitive types such as `int` or `char`. In no case should variables and functions be declared on the same line, e.g.,

```
long    dbaddr, get_dbaddr();   /* WRONG */
```

8.2. Indentation Within Declarations

Many programmers at Sun like to insert tabs in their variable declarations to align the variables. They feel this makes their code more readable. For example:

```
int             x;
extern int      y;
register int    count;
char            **pointer_to_string;
```

If variables with long type names are declared along with variables with short type names, it may be best to indent the variables one or two tab stops, and only use a single space after the long type names, e.g.,

11. Many editors include a command to find a matching brace, making it easy to navigate from the right brace back to the corresponding statement.

```
int             x, y, z;
extern int      i;
register int    count;
struct very_long_name *p;
```

It is also acceptable to use only a single space between the type name and the variable name:

```
int x;
char c;
enum rainbow y;
```

There is no Sun standard regarding how far over to indent variable names. The user should choose a value that looks pleasing to him, taking into consideration how frequently he employs lengthy declarations, but note that declarations such as the following probably make the code **harder** to read:

```
struct very_long_structure_name                 *p;
struct another_very_long_structure_name         *q;
char                                            *s;
int                                             i;
short                                           r;
```

Note that the use of `#define` to declare constants and macros follows indentation rules similar to those for other declarations. In particular, the `#define`, the macro name, and the macro text should all be separated from each other by tabs and properly aligned.

8.3. Local Declarations

Do not declare the same variable name in an inner block[12]. For example,

```
func()
{
        int cnt;
        ...
        if (condition) {
                register int cnt;       /* WRONG */
                ...
        }
        ...
}
```

Even though this is valid C, the potential confusion is enough that *lint* will complain about it when given the −**h** option.

8.4. External Declarations

External declarations should begin in column 1. Each declaration should be on a separate line. A comment describing the role of the object being declared should be included, with the exception that a list of defined constants does not need comments if the constant names are sufficient documentation. The comments should be tabbed so that they line up underneath each other[13]. Use the tab character rather than blanks. For structure and union template declarations, each element should be alone on a line with a comment describing it. The opening brace ({) should be on the same line as the structure tag, and the closing brace should be alone on a line in column 1, e.g.,

12. In fact, avoid any local declarations that override declarations at higher levels.

13. So should the constant names and their defined values.

```
struct  boat {
        int     wllength;       /* water line length in feet */
        int     type;           /* see below */
        long    sarea;          /* sail area in square feet */
};

/*
 * defines for boat.type
 */
#define KETCH   1
#define YAWL    2
#define SLOOP   3
#define SQRIG   4
#define MOTOR   5
```

In any file which is part of a larger whole rather than a self-contained program, maximum use should be made of the `static` keyword to make functions and variables local to single files. Variables in particular should be accessible from other files only when there is a clear need that cannot be filled in another way. Such usages should be commented to make it clear that another file's variables are being used.

8.5. Function Definitions

Each function should be preceded by a block comment prologue that gives the name and a short description of what the function does. The type of the value returned should be alone on a line in column 1 (`int` should be specified explicitly). If the function does not return a value then it should be given the return type `void`. If the value returned requires a long explanation, it should be given in the prologue. Functions that are not used outside of the file in which they are declared should be declared as `static`. This lets the reader know explicitly that a function is private, and also eliminates the possibility of name conflicts with variables and procedures in other files.

When defining functions using the old K&R syntax, the function name and formal parameters should be alone on a line beginning in column 1. This line is followed by the declarations of the formal parameters. Each parameter should be declared (do not default to `int`), and tabbed over one indentation level. The opening brace of the function body should also be alone on a line beginning in column 1.

For functions defined using the ANSI C syntax, the style is much the same. The examples below illustrate the acceptable forms of ANSI C declarations.

All local declarations and code within the function body should be tabbed over at least one tab. (Labels may appear in column 1.) If the function uses any external variables or functions that are not otherwise declared `extern`, these should have their own declarations in the function body using the `extern` keyword. If the external variable is an array, the array bounds must be repeated in the `extern` declaration.

If an external variable or a parameter of type pointer is changed by the function, that fact should be noted in the comment.

All comments about parameters and local variables should be tabbed so that they line up underneath each other. The declarations should be separated from the function's statements by a blank line.

The following examples illustrate many of the rules for function definitions.

```
/*
 * sky_is_blue()
 *
 * Return true if the sky is blue, else false.
 */
int
sky_is_blue()
{
        extern int hour;

        if (hour < MORNING || hour > EVENING)
                return (0);     /* black */
        else
                return (1);     /* blue */
}

/*
 * tail(nodep)
 *
 * Find the last element in the linked list
 * pointed to by nodep and return a pointer to it.
 */
Node *
tail(nodep)
        Node *nodep;
{
        register Node *np;      /* current pointer advances to NULL */
        register Node *lp;      /* last pointer follows np */

        np = lp = nodep;
        while ((np = np->next) != NULL)
                lp = np;
        return (lp);
}

/*
 * ANSI C Form 1.
 * Use this form when the arguments easily fit on one line,
 * and no per-argument comments are needed.
 */
int
foo(int alpha, char *beta, struct bar gamma)
{
        ...
}
```

```
/*
 * ANSI C Form 2.
 * This is a variation on form 1, using the standard continuation
 * line technique (indent by 4 spaces).  Use this form when no
 * per-argument comments are needed, but all argument declarations
 * won't fit on one line.  This form is generally frowned upon,
 * but acceptable.
 */
int
foo(int alpha, char *beta,
    struct bar gamma)
{
        ...
}

/*
 * ANSI C Form 3.
 * Use this form when per-argument comments are needed.
 * Note that each line of arguments is indented by a full
 * tab stop.  Note carefully the placement of the left
 * and right parentheses.
 */
int
foo(
        int alpha,              /* first arg */
        char *beta,             /* arg with a vert long comment needed */
                                /*   to describe its purpose */
        struct bar gamma)       /* big arg */
{
        ...
}
```

8.6. Type Declarations

Many programmers at Sun use named types, i.e., `typedef`s, liberally. They feel that the use of `typedef`s simplifies declaration lists and can make program modification easier when types must change. Other programmers feel that the use of a `typedef` hides the underlying type when they want to know what the type is. This is particularly true for programmers who need to be concerned with efficiency, e.g., kernel programmers, and therefore need to be aware of the implementation details. The choice of whether or not to use `typedef`s is left to the implementor.

It should be noted, however, that `typedef`s should be used to isolate implementation details, rather than to save keystrokes[14]. For instance, the following example demonstrates two inappropriate uses of `typedef`. In both cases, the code would pass *lint*[15], but nevertheless depends on the underlying types and would break if these were to change:

14. An exception is made in the cases of `u_char`, `u_short`, etc. that are defined in <*sys/types.h*>.

15. Some people claim that this is a bug in *lint*.

```
typedef char *Mything1;         /* These typedefs are inappropriately */
typedef int Mything2;           /*   used in the code that follows. */

int
can_read(t1)
        Mything1 t1;
{
        Mything2 t2;

        t2 = access(t1, R_OK);  /* access() expects a (char *) */
        if (t2 == -1)           /*   and returns an (int) */
                takeaction();
        return (t2);            /* can_read() returns an (int) */
}
```

If one elects to use a `typedef` in conjunction with a pointer type, the underlying type should be `typedef`-ed, rather than `typedef`-ing a pointer to underlying type, because it is often necessary and usually helpful to be able to tell if a type is a pointer. Thus, in the example in section 3, `Symbol` is defined to be a `struct symbol`, rather than `struct symbol *`, and `insert()` defined to return `Symbol *`.

9. Statements

Each line should contain at most one statement. In particular, do not use the comma operator to group multiple statements on one line, or to avoid using braces. For example,

```
argv++; argc--;         /* WRONG */

if (err)
        fprintf(stderr, "error"), exit(1);      /* VERY WRONG */
```

Do not nest the ternary conditional operator (?:). For example:

```
num = cnt < tcnt ? (cnt < fcnt ? fcnt : cnt) :
    tcnt < bcnt ? tcnt : bcnt > fcnt ? fcnt : bcnt;     /* WRONG */
```

If the `return` statement is used to return a value, the expression should always be enclosed in parentheses.

Functions that return no value should **not** include a `return` statement as the last statement in the function.

9.1. Compound Statements

Compound statements are statements that contain lists of statements enclosed in {} braces. The enclosed list should be indented one more level than the compound statement itself. The opening left brace should be at the end of the line beginning the compound statement and the closing right brace should be alone on a line, positioned under the beginning of the compound statement. (See examples below.) Note that the left brace that begins a function body is the only occurrence of a left brace which should be alone on a line.

Braces are also used around a single statement when it is part of a control structure, such as an `if-else` or `for` statement, as in:

```
if (condition) {
        if (other_condition)
                statement;
}
```

Some programmers feel that braces should be used to surround **all** statements that are part of control structures, even singletons, because this makes it easier to add or delete statements without thinking about whether braces should be added or removed[16]. Thus, they would write:

```
if (condition) {
        return (0);
}
```

In either case, if one arm of an `if-else` statement contains braces, all arms should contain braces. Also, if the body of a `for` or `while` loop is empty, no braces are needed:

```
while (*p++ != c)
        ;
```

9.2. Examples

if, if-else, if-else if-else statements

```
if (condition) {
        statements;
}


if (condition) {
        statements;
} else {
        statements;
}


if (condition) {
        statements;
} else if (condition) {
        statements;
}
```

Note that the right brace before the `else` and the right brace before the `while` of a `do-while` statement (see below) are the only places where a right brace appears that is not alone on a line.

for statements

```
for (initialization; condition; update) {
        statements;
}
```

When using the comma operator in the initialization or update clauses of a `for` statement, it is suggested that no more than three variables should be updated. More than this tends to make the expression too complex. In this case it is generally better to use separate statements outside the `for` loop (for the initialization clause), or at the end of the loop (for the update clause).

The infinite loop is written using a `for` loop.

```
for (;;) {
        statements;
```

16. Some programmers reason that, since some apparent function calls might actually be macros that expand into multiple statements, always using braces allows such macros to always work safely. Instead, we strongly discourage the use of such macros. If such macros must be used, they should be all upper case so as to clearly distinguish them as macros; see the Naming Conventions section.

```
	}
```

while statements

```
	while (condition) {
		statements;
	}
```

do-while statements

```
	do {
		statements;
	} while (condition);
```

switch statements

```
	switch (condition) {
	case ABC:
	case DEF:
		statements;
		break;
	case XYZ:
		statements;
		break;
	default:
		statements;
		break;
	}
```

The last `break`[17] is, strictly speaking, redundant, but it is recommended form nonetheless because it prevents a fall-through error if another `case` is added later after the last one. In general, the fall-through feature of the C `switch` statement should rarely, if ever, be used (except for multiple case labels as shown in the example). If it is, it should be commented for future maintenance.

All `switch` statements should include a default case[18]. Don't assume that the list of cases covers all possible cases. New, unanticipated, cases may be added later, or bugs elsewhere in the program may cause variables to take on unexpected values.

The `case` statement should be indented to the same level as the `switch` statement. The `case` statement should be on a line separate from the statements within the case.

The next example shows the format that should be used for a switch whenever the blocks of statements contain more than a couple of lines. Note the use of blank lines to set off the individual arms of the switch. Note also the use of a block local to one case to declare variables.

---

17. A `return` statement is sometimes substituted for the `break` statement, especially in the `default` case.

18. With the possible exception of a switch on an `enum` variable for which all possible values of the `enum` are listed.

```
switch (condition) {

case ABC:
case DEF:
        statement1;
        .
        .
        statementn;
        break;

case XYZ: {
        int var1;

        statement1;
        .
        .
        statementm;
        break;
}

default:
        statements;
        break;
}
```

10. Using White Space

10.1. Vertical White Space

The previous example illustrates the use of blank lines to improve the readability of a complicated switch statement. Blank lines improve readability by setting off sections of code that are logically related. Generally, the more white space in code (within reasonable limits), the more readable it is.

A blank line should always be used in the following circumstances:

- After the `#include` section.
- After blocks of `#define`s of constants, and before and after `#define`s of macros.
- Between structure declarations.
- Between procedures.
- After local variable declarations.

Form-feeds should never be used to separate functions. Instead, separate functions into separate files, if desired.

10.2. Horizontal White Space

Here are the guidelines for blank spaces:

- A blank should follow a keyword[19] whenever a parenthesis follows the keyword. Blanks should not be used between procedure names (or macro calls) and their argument list. This helps to distinguish keywords from procedure calls.

19. Note that `sizeof` and `return` are keywords, whereas `strlen` and `exit` are not.

```
        if (strcmp(x, "done") == 0)     /* no space between strcmp and '(' */
                return (0);             /* space between return and '(' */
```

- Blanks should appear after the commas in argument lists.
- Blanks should **not** appear immediately after a left parenthesis or immediately before a right parenthesis.
- All binary operators except `.` and `->` should be separated from their operands by blanks[20]. In other words, blanks should appear around assignment, arithmetic, relational, and logical operators. Blanks should never separate unary operators such as unary minus, address (‘`&`’), indirection (‘`*`’), increment (‘`++`’), and decrement (‘`--`’) from their operands. Note that this includes the unary `*` that is a part of pointer declarations.

Examples:

```
        a += c + d;
        a = (a + b) / (c * d);
        strp->field = str.fl - ((x & MASK) >> DISP);
        while (*d++ = *s++)
                n++;
```

- The expressions in a `for` statement should be separated by blanks, e.g.,

```
        for (expr1; expr2; expr3)
```

- Casts should not be followed by a blank, with the exception of function calls whose return values are ignored, e.g.,

```
        (void) myfunc((unsigned)ptr, (char *)x);
```

10.3. Hidden White Space

There are many uses of blanks that will not be visible when viewed on a terminal, and it is often difficult to distinguish blanks from tabs. However, inconsistent use of blanks and tabs may produce unexpected results when the code is printed with a pretty-printer, and may make simple regular expression searches fail unexpectedly. The following guidelines are helpful:

- Avoid spaces and tabs at the end of a line.
- Avoid spaces between tabs and tabs between spaces.
- Use tabs to line things up in columns (e.g., for indenting code, and to line up elements within a series of declarations) and spaces to separate items within a line.
- Use tabs to separate single line comments from the corresponding code.

11. Parenthesization

Since C has some unexpected precedence rules, it is generally a good idea to use parentheses liberally in expressions involving mixed operators. It is also important to remember that complex expressions can be used as parameters to macros, and operator-precedence problems can arise unless **all** occurrences of parameters in the body of a macro definition have parentheses around them.

12. Naming Conventions

Identifier conventions can make programs more understandable by making them easier to read. They can also give information about the function of the identifier, e.g., constant, named type, that can be helpful in understanding code. Individual projects will no doubt have their own naming conventions.

20. Some judgment is called for in the case of complex expressions, which may be clearer if the ‘‘inner’’ operators are not surrounded by spaces and the ‘‘outer’’ ones are.

However, each programmer should be consistent about his use of naming conventions.

Here are some general rules about naming.

- Variable and function names should be short yet meaningful. One character variable names should be avoided except for temporary ‘‘throwaway’’ variables. Use variables `i`, `j`, `k`, `m`, `n` for integers, `c`, `d`, `e` for characters, `p`, `q` for pointers, and `s`, `t` for character pointers. Avoid variable `l` because it is hard to distinguish `l` from `1` on some printers and displays.
- Pointer variables should have a ‘‘p’’ appended to their names for each level of indirection. For example, a pointer to the variable `dbaddr` (which contains disk block addresses) can be named `dbaddrp` (or perhaps simply `dp`). Similarly, `dbaddrpp` would be a pointer to a pointer to the variable `dbaddr`.
- Separate "words" in a long variable name with underscores, e.g., `create_panel_item`[21]. An initial underscore should never be used in any user-program names[22]. Trailing underscores should be avoided too.
- `#define` names for constants should be in all CAPS.
- Two conventions are used for named types, i.e., `typedef`s. Within the kernel named types are given a name ending in `_t`, e.g.,

  ```
  typedef enum { FALSE, TRUE } bool_t;
  typedef struct node node_t;
  ```

  In many user programs named types have their first letter capitalized, e.g.,

  ```
  typedef enum { FALSE, TRUE } Bool;
  typedef struct node Node;
  ```

- Macro names may be all CAPS or all lower case. Some macros (such as *getchar* and *putchar*) are in lower case since they may also exist as functions. There is a slight preference for all upper case macro names.
- Variable names, structure tag names, and function names should be in lower case.

Note: in general, with the exception of named types, it is best to avoid names that differ only in case, like `foo` and `FOO`. The potential for confusion is considerable. However, it is acceptable to use as a typedef a name which differs only in capitalization from its base type, e.g.,

```
typedef struct node Node;
```

It is also acceptable to give a variable of this type a name that is the all lower case version of the type name, e.g.,

```
Node node;
```

- The individual items of enums should be guaranteed unique names by prefixing them with a tag identifying the package to which they belong. For example,

  ```
  enum rainbow { RB_red, RB_orange, RB_yellow, RB_green, RB_blue };
  ```

  The *dbx* debugger supports enums in that it can print out the value of an enum, and can also perform assignment statements using an item in the range of an enum. Thus, the use of enums over equivalent `#define`s may make program debugging easier. For example, rather than writing:

  ```
  #define SUNDAY  0
  #define MONDAY  1
  ```

  write:

21. Mixed case names, e.g., `CreatePanelItem`, are strongly discouraged.
22. Initial underscores are reserved for global names that are internal to software library packages.

```
    enum day_of_week { dw_sunday, dw_monday, ... };
```

- Implementors of libraries should take care to hide the names of any variables and functions that have been declared `extern` because they are shared by several modules in the library, but nevertheless are private to the library. One technique for doing this is to prefix the name with an underscore and a tag that is unique to the package, e.g., `_panel_caret_mpr`.

13. Continuation Lines

Occasionally, an expression will not fit in the available space in a line, for example, a procedure call with many arguments, or a conjunction or disjunction with many arms. Such occurrences are especially likely when blocks are nested deeply or long identifiers are used. If this happens, the expression should be broken after the last comma in the case of a function call (never in the middle of a parameter expression), or after the last operator that fits on the line (the continuation line should never start with a binary operator). The next line should be further indented by half a tab stop. If they are needed, subsequent continuation lines should be broken in the same manner, and aligned with each other. For example,

```
if (long_logical_test_1 || long_logical_test_2 ||
    long_logical_test_3) {
        statements;
}


a = (long_identifier_term1 - long_identifier_term2) *
     long_identifier_term3;


function(long_complicated_expression1, long_complicated_expression2,
    long_complicated_expression3, long_complicated_expression4,
    long_complicated_expression5, long_complicated_expression6)
```

14. Constants

Numerical constants should not be coded directly. The `#define` feature of the C preprocessor should be used to assign a meaningful name. This will also make it easier to administer large programs since the constant value can be changed uniformly by changing only the `#define`. The enum data type is the preferred way to handle situations where a variable takes on only a discrete set of values, since additional type checking is available through *lint*. As mentioned above, the *dbx* debugger also provides support for enums.

There are some cases where the constants 0 and 1 may appear as themselves instead of as `#define`s. For example if a `for` loop indexes through an array, then

```
for (i = 0; i < ARYBOUND; i++)
```

is reasonable while the code

```
fptr = fopen(filename, "r");
if (fptr == 0)
        error("can’t open %s\n", filename);
```

is not. In the last example, the defined constant `NULL` is available as part of the standard I/O library’s header file <*stdio.h*>, and should be used in place of the 0.

In rare cases, other constants may appear as themselves. Some judgement is required to determine whether the semantic meaning of the constant is obvious from its value, or whether the code would be easier to understand if a symbolic name were used for the value.

15. Goto

While not completely avoidable, use of `goto` is discouraged. In many cases, breaking a procedure into smaller pieces, or using a different language construct will enable elimination of `goto`s. For example, instead of:

```
again:
        if (s = proc(args))
                if (s == -1 && errno == EINTR)
                        goto again;
```

write:

```
        do {
                s = proc(args);
        } while (s == -1 && errno == EINTR);    /* note[23] */
```

The main place where `goto`s can be usefully employed is to break out of several levels of `switch`, `for`, and `while` nesting[24], e.g.,

```
        for (...)
                for (...) {
                        ...
                        if (disaster)
                                goto error;
                }
        ...
error:
        clean up the mess
```

Never use a `goto` outside of a given block to branch to a label within a block:

```
      goto label;     /* WRONG */
      ...
      for (...) {
              ...
      label:
              statement;
              ...
      }
```

When a `goto` is necessary, the accompanying label should be alone on a line and positioned one indentation level to the left of the code that follows. If a label is not followed by a program statement (e.g., if the next token is a closing brace (})) a NULL statement (;) must follow the label.

16. Variable Initialization

C permits initializing a variable where it is declared. Programmers at Sun are equally divided about whether or not this is a good idea:

> "I like to think of declarations and executable code as separate units. Intermixing them only confuses the issue. If only a scattered few declarations are initialized, it is easy not to see them."
>
> "The major purpose of code style is clarity. I think the less hunting around for the connections between different places in the code, the better. I don't think variables should be

23. These two expressions are equivalent unless `s` is asynchronously modified, e.g., if `s` is an I/O register.

24. The need to do such a thing may indicate that the inner constructs should be broken out into a separate function, with a success/failure return code.

initialized for no reason, however. If the variable doesn't need to be initialized, don't waste the reader's time by making him/her think that it does."

A convention used by some programmers is to only initialize automatic variables in declarations if the value of the variable is constant throughout the block.

The decision about whether or not to initialize a variable in a declaration is therefore left to the implementor. Use good taste. For example, don't bury a variable initialization in the middle of a long declaration, e.g.,

```
int     a, b, c, d = 4, e, f;           /* This is NOT good style */
```

17. Multiple Assignments

C also permits assigning several variables to the same value in a single statement, e.g.,

```
x = y = z = 0;
```

Good taste is required here also. For example, assigning several variables that are used the same way in the program in a single statement clarifies the relationship between the variables by making it more explicit, e.g.,

```
x = y = z = 0;
vx = vy = vz = 1;
count = 0;
scale = 1;
```

is good, whereas:

```
x = y = z = count = 0;
vx = vy = vz = scale = 1;
```

sacrifices clarity for brevity. In any case, the variables that are so assigned should all be of the same type (or all pointers being initialized to `NULL`). It is not a good idea (because it is hard to read) to use multiple assignments for complex expressions, e.g.,

```
foo_bar.fb_name.firstchar = bar_foo.fb_name.lastchar = 'c';     /* Yecch */
```

18. Preprocessor

Do not rename members of a structure using `#define` within a subsystem; instead, use a *union*. However, `#define` can be used to define shorthand notations for referencing members of a union. For example, instead of

```
struct proc {
        ...
        int     p_lock;
        ...
};
...
in a subsystem:
#define p_label p_lock
```

use

```
struct proc {
        ...
        union {
                int     p_Lock;
                int     p_Label;
        } p_un;
        ...
};
#define p_lock  p_un.p_Lock
#define p_label p_un.p_Label
```

Be *extremely* careful when choosing names for `#define`s. For example, never use something like

```
#define size    10
```

especially in a header file, since it is not unlikely that the user might want to declare a variable named `size`.

Remember that names used in `#define` statements come out of a global preprocessor name space and can conflict with names in any other namespace. For this reason, this use of `#define` is discouraged.

Note that `#define` follows indentation rules similar to other declarations; see the section on indentation for details.

Care is needed when defining macros that replace functions since functions pass their parameters by value whereas macros pass their arguments by name substitution.

At the end of an `#ifdef` construct used to select among a required set of options (such as machine types), include a final `#else` clause containing a useful but illegal statement so that the compiler will generate an error message if none of the options has been defined:

```
#ifdef vax
        ...
#elif sun
        ...
#elif u3b2
        ...
#else
#error unknown machine type;
#endif /* machine type */
```

Note that `#elif` is an ANSI C construct and should not be used in header files that must be able to be processed be older K&R C compilers. Note also the use of the ANSI C `#error` statement, which also has the desired effect when using a K&R C compiler.

Don't change C syntax via macro substitution, e.g.,

```
#define BEGIN   {
```

It makes the program unintelligible to all but the perpetrator.

19. Miscellaneous Comments on Good Taste

Try to make the structure of your program match the intent. For example, replace:

```
if (boolean_expression)
        return (TRUE);
else
        return (FALSE);
```

with:

```
	return (boolean_expression);
```

Similarly,

```
	if (condition)
		return (x);
	return (y);
```

is usually clearer when written as:

```
	if (condition)
		return (x);
	else
		return (y);
```

or even better, if the condition and return expressions are short;

```
	return (condition ? x : y);
```

Do not default the boolean test for nonzero, i.e.

```
	if (f() != FAIL)
```

is better than

```
	if (f())
```

even though `FAIL` may have the value 0 which is considered to mean false by C[25]. This will help you out later when somebody decides that a failure return should be −1 instead of 0. An exception is commonly made for predicates, which are functions which meet the following restrictions:

- Has no other purpose than to return true or false.
- Returns 0 for false, non-zero for true.
- Is named so that the meaning of (say) a 'true' return is absolutely obvious. Call a predicate `is_valid` or `valid`, not `check_valid`.

Never use the boolean negation operator (`!`) with non-boolean expressions. In particular, never use it to test for a NULL pointer or to test for success of the `strcmp` function, e.g.,

```
	char *p;
	...
	if (!p)			/* WRONG */
		return;

	if (!strcmp(*argv, "-a"))	/* WRONG */
		aflag++;
```

Do not use the assignment operator in a place where it could be easily confused with the equality operator For instance, in the simple expression

```
	if (x = y)
		statement;
```

it is hard to tell whether the programmer really meant assignment or the equality test. Instead, use

```
	if ((x = y) != 0)
		statement;
```

or something similar if the assignment is needed within the `if` statement.

25. A particularly notorious case is using `strcmp` to test for string equality, where the result should never be defaulted.

There is a time and a place for embedded assignments[26]. In some constructs there is no better way to accomplish the results without making the code bulkier and less readable. For example:

```
while ((c = getchar()) != EOF) {
        process the character
}
```

Using embedded assignments to improve run-time performance is also possible. However, one should consider the tradeoff between increased speed and decreased maintainability that results when embedded assignments are used in artificial places. For example, the code:

```
a = b + c;
d = a + r;
```

should not be replaced by

```
d = (a = b + c) + r;
```

even though the latter may save one cycle. Note that in the long run the time difference between the two will decrease as the optimizer gains maturity, while the difference in ease of maintenance will increase as the human memory of what's going on in the latter piece of code begins to fade[27].

There is also a time and place for the ternary `? :` operator and the binary comma operator. If an expression containing a binary operator appears before the `?`, it should be parenthesized:

```
(x >= 0) ? x : -x
```

Nested `? :` operators can be confusing and should be avoided if possible. There are some macros like *getchar* where they can be useful. The comma operator can also be useful in `for` statements to provide multiple initializations or incrementations.

20. Portability

The advantages of portable code are well known. This section gives some guidelines for writing portable code, where the definition of portable is a source file can be compiled and executed on different machines with the only source change being the inclusion of (possibly different) header files. The header files will contain `#define`s and `typedef`s that may vary from machine to machine.

There are two aspects of portability that must be considered: hardware compatibility and interface compatibility. The former category encompasses problems arising from differing machine types, e.g., byte-ordering differences. The second category deals with the multiplicity of UNIX operating system interfaces, e.g., BSD4.3 and System V. As the POSIX standard (IEEE P1003.1 Portable Operating System Interface) becomes widely used, it will be preferable to write programs and software packages that use POSIX semantics exclusively. Where this is not possible (for example, POSIX does not define the BSD *socket* interface), the required extensions should be documented in a comment near the top of the source file and/or in accompanying manuals[28].

The POSIX standards provide for portability across a wide range of systems. The X/Open XPG standards are emerging as important standards for portability across a wide range of systems. Generally it is best to avoid vendor-specific extensions and use the standard interface that best matches your requirements for portability, performance, functionality, etc.

The following is a list of pitfalls to be avoided and recommendations to be considered when designing portable code:

- First, one must recognize that some things are inherently non-portable. Examples are code to deal with particular hardware registers such as the program status word, and code that is designed to

26. The ++ and −− operators count as assignments. So, for many purposes, do functions with side effects.

27. Note also that side effects within expressions can result in code whose semantics are compiler-dependent, since C's order of evaluation is explicitly undefined in most places. Compilers do differ.

28. Stringent documentation requirements are defined for programs that claim POSIX conformance.

support a particular piece of hardware such as an assembler or I/O driver. Even in these cases there are many routines and data organizations that can be made machine-independent. Source files should be organized so that the machine-independent code and the machine-dependent code are in separate files. Then if the program is to be moved to a new machine, it is a much easier task to determine what needs to be changed[29]. It is also possible that code in the machine-independent files may have uses in other programs as well.

• Pay attention to word sizes. The following sizes apply to basic types in C for some common machines:

type	PDP11	VAX	Mac	PC/DOS	680x0	SPARC	PC/UNIX
char	8	8	8	8	8	8	8
short	16	16	16	16	16	16	16
int	16	32	16	16	32	32	32
long	32	32	32	32	32	32	32
pointer	16	32	32	16	32	32	32

In general, if the word size is important, `short` or `long` should be used to get 16- or 32-bit items on any of the above machines[30]. If a simple loop counter is being used where either 16 or 32 bits will do, then use `int`, since it will get the most efficient (natural) unit for the current machine.

• Word size also affects shifts and masks. The code

```
x &= 0177770
```

will clear only the three rightmost bits of an *int* on a DOS PC. On a SPARC it will also clear the entire upper halfword. Use

```
x &= ~07
```

instead which works properly on all machines[31].

• Beware of making assumptions about the size of pointers. They are not always the same size as `int`[32]. Also, be aware of potential pointer alignment problems. On machines that do not support a uniform address space (unlike, e.g., SPARC), the conversion of a pointer-to-character to a pointer-to-int may result in an invalid address.

• Watch out for signed characters. On the SPARC, characters are sign extended when used in expressions, which is not the case on some other machines. In particular, *getchar* is an integer-valued function (or macro) since the value of `EOF` for the standard I/O library is -1, which is not possible for a character on the IBM 370[33]. If the code depends on the character being signed rather than unsigned, it's probably best to use the ANSI C `signed` keyword.

• On some processors on which C exists the bits (or bytes) are numbered from right to left within a word. Other machines number the bits from left to right. Hence any code that depends on the left-right orientation of bits in a word deserves special scrutiny. Bit fields within structure members will only be portable so long as two separate fields are never concatenated and treated as

29. If you `#ifdef` dependencies, make sure that if no machine is specified, the result is a syntax error, **not** a default machine!

30. Any unsigned type other than plain `unsigned int` should be `typedef`ed, as such types are highly compiler-dependent. This is also true of long and short types other than `long int` and `short int`. Large programs should have a central header file which supplies `typedef`s for commonly used width-sensitive types, to make it easier to change them and to aid in finding width-sensitive code.

31. The or operator (|) does not have these problems, nor do bitfields.

32. Nor are all pointers always the same size, or freely interchangeable. Pointer-to-character is a particular trouble spot on machines that do not address to the byte.

33. Actually, this is not quite the real reason why *getchar* returns `int`, but the comment is valid: code that assumes either that characters are signed or that they are unsigned is unportable. It is best to completely avoid using `char` to hold numbers. Manipulation of characters as if they were numbers is also often unportable.

a unit. The same applies to variables in general. Alignment considerations and loader peculiarities make it very rash to assume that two consecutively declared variables are together in memory, or that a variable of one type is aligned appropriately to be used as another type.

- Become familiar with existing library functions and `#define`s[34]. You should not be writing your own string compare routine, or making your own `#define`s for system structures[35]. Not only does this waste your time, but it prevents your program from taking advantage of any microcode assists or other means of improving performance of system routines[36].
- Use *lint* and *make* (see next sections).

21. Lint

Lint is a C program checker that examines C source files to detect and report type incompatibilities, inconsistencies between function definitions and calls, potential program bugs, etc. It is a good idea to use *lint* on programs that are being released to a wide audience.

It should be noted that the best way to use *lint* is not as a barrier that must be overcome before official acceptance of a program, but rather as a tool to use whenever major changes or additions to the code have been made. *Lint* can find obscure bugs and insure portability before problems occur.

22. Make

Make is a program that interprets a description file (`Makefile`) in order to produce *shell* commands that generate target files from their sources. All projects should have a `Makefile` in the top-level source directory that contains rules to build all of the relevant targets. In general, the top-level `Makefile` will be very simple, containing rules that invoke recursive *make*s in the sub-directories.

In addition to project-dependent targets, the `Makefile` should contain rules to build the following targets:

`all`	builds all targets
`install`	installs all targets and header files in the appropriate directories
`clean`	removes all intermediate files
`clobber`	removes all targets and intermediate files
`lint`	executes *lint* on all targets

For SunOS, more detailed Makefile guidelines are specified elsewhere.

23. Project-Dependent Standards

Individual projects may wish to establish additional standards beyond those given here. The following issues are some of those that should be addressed by projects.

- What additional naming conventions should be followed? In particular, systematic prefix conventions for functional grouping of global data and also for structure or union member names can be useful.
- What kind of include file organization is appropriate for the project's particular data hierarchy?
- What procedures should be established for reviewing *lint* complaints? A tolerance level needs to be established in concert with the *lint* options to prevent unimportant complaints from hiding complaints about real bugs or inconsistencies.

---

34. But not **too** familiar. The internal details of library facilities, as opposed to their external interfaces, are subject to change without warning. They are also often quite unportable.

35. Or, especially, writing your own code to control terminals. Use the *termcap*, *terminfo*, or *curses* packages.

36. It also makes your code less readable, because the reader has to figure out whether you're doing something special in the reimplemented stuff to justify its existence. Furthermore, it's a fruitful source of bugs.

- If a project establishes its own archive libraries, it should plan on supplying a *lint* library file to the system administrators. This will allow *lint* to check for compatible use of library functions.

24. Conclusion

A set of standards has been presented for C programming style. One of the most important points is the proper use of white space and comments so that the structure of the program is evident from the layout of the code. Another good idea to keep in mind when writing code is that it is likely that you or someone else will be asked to modify it or make it run on a different machine some time in the future.

Bibliography

[1] S. Shah, *C Coding Guidelines for UNIX System Development Issue 3*, AT&T (internal document) 1987.

[2] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall 1978.

[3] S.P. Harbison and G.L. Steele, *A C Reference Manual*, Prentice-Hall 1984.

[4] Evan Adams, Dave Goldberg, et al, *Naming Conventions for Software Packages*, Sun Microsystems (internal document) 1987.

[5] ANSI/X3.159-198x, *Programming Language C Standard*, (Draft) 1986.

[6] IEEE/P1003.1, *Portable Operating System for Computer Environments*, (Draft) 1987.

25. Appendix 1: SCCS ident strings and copyrights

The following are SCCS ident strings and copyrights for various kinds of files. (Omit the Copyright comment if not relevant.) Note that `%W%` is an acceptable substitute for `%Z%%M% %I%`. Note also that there are tabs between the `%M%`, `%I%`, `%E%`, and `%W%` keywords, to make the output of the **what(1)** command more readable.

Header Files

```
/*
 * Copyright (c) 1993 by Sun Microsystems, Inc.
 * All rights reserved.
 */

#ifndef guard
#define guard

#pragma ident   "%Z%%M% %I% %E% SMI"

#ifdef  __cplusplus
extern "C" {
#endif

body of header

#ifdef  __cplusplus
}
#endif

#endif guard
```

C or Assembler Files

```
/*
 * Copyright (c) 1993 by Sun Microsystems, Inc.
 * All rights reserved.
 */

#pragma ident   "%Z%%M% %I% %E% SMI"
```

Makefiles

```
#
# %Z%%M% %I% %E% SMI
#
```

Shell Files

```
#!/bin/sh       (or /bin/ksh, or /bin/csh)
#
# %Z%%M% %I% %E% SMI
#
```

Manual Pages

```
 .\" %Z%%M% %I% %E% SMI
```