C Style and Coding Standards

Glenn Skinner
Suryakanta Shah
Bill Shannon

AT&T Information System
Sun Microsystems

ABSTRACT

This document describes a set of coding standards and recommendations for programs written in the C language at AT&T and Sun Microsystems. The purpose of having these standards is to facilitate sharing of each other's code, as well as to enable construction of tools (e.g., editors, formatters). Through the use of these tools, programmers will be helped in the development of their 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.

C Style and Coding Standards

Glenn Skinner
Suryakanta Shah
Bill Shannon

AT&T Information System
Sun Microsystems

1. Introduction

The scope of this document is the coding style used at AT&T and Sun in writing C programs. A common coding style makes it easier for several people to cooperate in the development of the same program. Using uniform coding style to develop systems will improve readability and facilitate maintenance. In addition, it will enable the construction of tools that incorporate knowledge of these standards to help programmers in the development of programs.

For certain style issues, such as the number of spaces used for indentation and the format of variable declarations, no clear consensus exists. In these cases, we have documented the various styles that are most frequently used. We strongly recommend, however, that within a particular project, and certainly within a package or module, only one style be employed. More important than the particular coding style used is consistency of coding style. 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.

These standards do not cover all situations. Experience and informed judgement should also be used. 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]

The following suffixes are required:

- c source file names end with a suffix of *.c*
- assembler source file names end with a suffix of *.s*[2]
- relocatable object file names end with a suffix of *.o*
- include header file names end with a suffix of *.h*
- archived library files end with a suffix of *.a*
- c-shell source file names end with a suffix of *.csh*
- bourne shell source file names end with a suffix of *.sh*
- yacc source file names end with a suffix of *.y*
- lex source file names end with a suffix of *.l*

---

1. In addition to the suffix convention given here, it is conventional to provide "makefile" for the control file for *make* and 'README' for a summary of the contents of a directory or directory tree.
2. Assembler source files that must be pre-processed (using *cpp*) may end with a suffix of *.S*

3. Program Organization

A modular code with well-defined interface is less prone to software errors, easier to maintain, and easier to enhance. Although C does not provide mechanisms for defining the modules of a program, modularity can be achieved using the techniques and conventions described below.

A module is a group of procedures and data that together implement some abstraction, for example, a memory management, symbol table, etc. Procedures and data 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 declaration 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(.h file). The code for the procedures and data definition is provided by the implementation file (.c file). Private definitions in the implementation file are declared to be **static**. See Appendix A for an example.

Provide **modular** code where practical. The extent of modularity depends on the complexity of the problem and the performance trade-off. Use discretion in determining an appropriate balance between the two. You might want to use preprocessor macros for simple operations.

4. File Organization

Sections in a file should be separated by blank lines. Although there is no maximum size requirement for source files, avoid using more than 3000 lines because they are cumbersome to deal with. Also, lines longer than 80 columns should be avoided because they are not handled well by all terminals[3].

4.1 Header Files

The order of sections for a *.h* file (interface file) is as follows:

1. **SCCS Identification**: The file should begin with the SCCS identification (copyright). See Appendix B for SCCS conventions.
2. **#ifndef**: It is often convenient to nest header files, for example, to provide the user with a single header file which includes a number of other header files needed for a particular application in a given order. Nested header files, however, could cause a header file to be included more than once. A standard way to avoid inclusion of the same header file twice is to define a variable whose name consists of the name of the header file with illegal characters replaced by underscores, and prefixed by an underscore. Then, bracket the entire header file in an #ifndef statement, which will check whether that variable has been defined previously. A #ifndef should follow the SCCS id.
3. **Block Comment**: Immediately after the #ifndef should be a block comment describing the purpose of the objects in the files (whether they be functions, external data declarations, or something else). Keep the description short and to the point.
4. **#includes**: Any header file #includes should follow block comment.
5. **#defines**: Any #defines that apply to the module, but not to an element of a structure are next.
6. **typedefs**: Any typedefs should follow #defines.
7. **Structure Declarations**: If a set of #defines applies to a particular piece of global data (such as a flags word), the #defines should be immediately after the data declaration.

---

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

8. **Function Declarations**: Declarations of functions should follow structure declarations. All external functions should be declared, even those that return int, or do not return a value (declare them to return void).

9. **External Variable Declarations**: Any external variable declarations should be next.

10. Do **not** define static and global variables in a header file.[4]

4.2 Implementation Files

The order of sections for a *.c* file (implementation file) is as follows:

1. **SCCS Identification**: The file should begin with the SCCS identification (copyright).

2. **Block Comment**: Immediately after the SCCS id should be a block comment describing the contents of the file.

3. **#includes**: Any header file #includes should be next.

4. **typedefs and #defines**: Any typedefs and #define that apply to the file should follow #includes.

5. **Structure Definition**: The structure definition should be next.

6. **Global Variable**: They should follow structure definition.

7. **Procedure Declarations**: The procedures in the module should be declared next. Only the private procedures need to be declared. The public procedures are always declared in the corresponding header file that has been included. Although the C compiler does not require prior declaration of a procedure that returns an int, nevertheless it is considered good practice for all of the procedures defined in the file to be pre-declared here.

8. **Definitions**: Finally 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[5], 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 using alphabetical order.

5. Indentation

1. Each level should be indented by a **tab** (8 column tab).

2. A line should be limited to a page boundary (code should not wrap around the terminal page). In case of a wrap around, it is often the case that the nesting structure is too complex and the code would be clearer if it were rewritten.

3. Insert tabs in variable declarations to align the variables. This makes the code more readable. For example,

```
extern int      y;
register int    count;
char            **ptr_to_strpp;
```

---

4. Defining variables in a header file is often a symptom of poor partitioning of code between files.

5. 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.

4. **Function** name and formal parameters should be alone on a line in column one. It will make searching easier. The type of the value returned should be alone on a line in column one (**int** should be specified explicitly). Declarations of the formal parameters should follow function definition. Each parameter should be declared (do not default to **int**), and tabbed over one indentation level. The opening and closing braces of the function body should also be alone on a line in column one. All local declarations and code within the function body should be tabbed over at least one tab. (Labels may appear in column one.) The declarations should be separated from the function's statements by a blank line. For examples,

```
/*
 * This function finds 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);
}
```

5. Occasionally, an expression will not fit in the available space in a line, for example, a procedure call with many arguments. 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. In the case of function call, function argument in the next line should be aligned with the first character to the right of the left parenthesis in the previous line. For example,

```
function(long_expression1, long_expression2,
         long_expression3, long_expression4,
         long_expression5, long_expression6)
```

In other cases, the next line should be indented by a half or a full tab stop. If 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;
}
```

6. The rules on how to indent particular C constructs such as **if** statements, **for** statements, **switch** statements, etc., are described below in the section 7.2 on compound statements.

6. Comments in C Programs

Comments should only contain information that is germane to reading and understanding the program. Misleading comments are worse than no comments. Comments should include the following:

1. Nontrivial design decisions and out of ordinary situation such as "side effect" which help reader should be well commented. Avoid duplicating information that is obvious from the code.
2. If an external variable or a parameter of type pointer is changed by the function, that fact should be noted in a comment.
3. Provide appropriate comments if you are falling through a case statement.
4. Each declaration should be commented as to its use. Temporary "throwaway" (e.g., loop counter) variables and the self-evident names of constants are exception to this rule. The comments should be tabbed so that they line up underneath each other[6].
5. Avoid information that is likely to become out-of-date.
6. Comments should not be enclosed in large boxes drawn with asterisks or other characters.
7. Comments should never include special characters, such as form-feed and backspace.
8. A description of how the corresponding package is built or in what directory it should reside should not be included in a comment.
9. Comments should not include a list of authors nor a modification history.

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

6.1 Block Comments

Block comments are used to describe the contents of files, the functions of procedures, and data structures and algorithms.

1. Block comments should be used at the beginning of each file and before each procedure.
2. The file containing main() should include a description of the program functionality and its command line syntax at the beginning of the file. All other files should describe only their own contents.
3. The block comment that precedes each procedure should document its function, input parameters, algorithm, and returned value.

The opening /* of a block comment 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.

---

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

```
/*
 * 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.
 */
```

In many cases, block comments inside a function are appropriate. The format for such comments should follow the above convention except that they should be indented to the same indentation level as the code that they describe.

6.2 Single-Line Comments

Single-Line comments are useful to describe non-obvious loop conditions and return values of functions.

1. A single line comment must be indented over to the indentation level of the code that follows. The comment text should be separated from the opening /* and closing */ by a space[7].

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

2. A block comment should be used when a single line is insufficient.

6.3 Trailing Comments

Trailing comments are most useful for documenting declarations. Trailing comments appear on the same line as the code they describe.

- Trailing comments should be tabbed over far enough to separate them from the statements.
- All comments about parameters and local variables should be tabbed so that they line up underneath each other.
- If more than one trailing 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));  /* odd numbers only */
```

- Avoid the assembly language style of commenting every line of executable code with a trailing comment.

7. 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*/**.

7. **Declaration**

Following standards should be used for declaration:

1. One declaration per line is preferred, because it encourages commenting. For example, *this style*

```
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 acceptable for declaration of several temporary variables of primitive types such as int or char.

2. Do not mix variables and functions in a declaration statement. For example,

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

3. Do not declare the same variable name in an inner block[8]. For example,

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

   Even though this is valid C, the potential confusion is enough that *lint*[9] will complain about it when given the **–h** option.

4. 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 one. For example,

---

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

9. *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.

```
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
```

5. All **extern** variables that are defined in a given module, but that need to be referenced in other modules, must be declared in a separate header file. For example, the audit module can define the variable "aud_stat" by providing it in the "aud_ex.h" header file. Other modules which need to reference the "aud_stat" variable should include the "aud_ex.h" header file. Any module that needs to have **extern** variables must provide a modulename_ex.h header file.

6. Variables and functions whose access is confined to a single source file should be declared **static**[10]. This lets the reader know explicitly that a function and variable are private, and also eliminates the possibility of name conflicts with variables and procedures in other files.

7. Every function should be given an explicit return value. If the function does not return a value, then it should be given the return type **void**.

8. The **typedef** declarations are used to parameterize the program for portability and modification. It is also used to provide better documentation. The use of **typedef**, however, *hides* the underlying type. This is particularly a problem for programmers, e.g., kernel programmers, who need to be concerned with efficiency and therefore need to be aware of the implementation details. The choice of whether or not to use typedefs is left to the implementor. The typedefs should only be used for the reasons specified above, rather than to save keystrokes. The example below demonstrates two inappropriate uses of typedef[11].

```
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) */
}
```

10. Unfortunately, sometimes this is not desirable because it poses a problem of accessing these variables via a debugger.

11. In both cases, the code would pass *lint*, but depends on the underlying types and would break if these were to change: Some people claim that this is a bug in *lint*.

9. **Typedef** should not be used to define a pointer to a structure, because it is often necessary and usually helpful to be able to tell if a type is a pointer. For example,

```
typedef label {
      ...
      ...
} *seclebp;       /*WRONG*/
```

should not be used.

8. Statements

8.1 Simple Statements

1. Each line should contain at most one statement. For example,

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

2. Do not use the comma operator to group multiple statements on one line, or to avoid using braces. As illustrated below:

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

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

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

9. Compound Statements

Compound statements are statements that contain lists of statements enclosed in {} braces.

1. The enclosed list should be indented one more level in the compound statement itself.
2. 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[12]. (See examples below.)
3. To use braces around a single statement when it is part of a control structure, such as an **if-else** or **for** statement, is up to the individual. For example,

---

12. 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.

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

However, a particular project should adopt only one convention[13].

4. Do not use braces, if the body of a **for** or **while** loop is empty.

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

5. **if, if-else, if-else if-else** statements should have the following form:

```
if (condition) {
      statements;
}

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

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

6. **for statement** should have the following form:

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

When using the comma operator in the initialization or update clauses of a **for** statement, avoid more than three variables. More than three variables make the expression too complex. If needed, use separate statements outside the **for** loop (for the initialization clause), or at the end of the loop (for the update clause).

---

13. Braces makes it easier to add or delete statements without thinking about whether braces should be added or removed. Also function calls might actually be macros that expand into multiple statements, always using braces allows such macros to always work safely.

7. **while statement** should have the following form:

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

8. **do-while statement** should have the following form:

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

9. **switch statement** should have the following form:

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

Every **switch** statement should include a default case. The last **break**[14] is redundant, but it prevents a fall-through error if another **case** is added later after the last one. 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).

Whenever the blocks contain several statements, use a blank line to set off the individual arms of the switch. For example,

14. A **return** statement is sometimes substituted for the **break** statement, especially in the **default** case.

```
switch (condition) {

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

case XYZ:
      statement1;
      .
      .
      statementm;
      break;

default:
      statements;
      break;
}
```

10. White Space

10.1 Blank Lines

Blank lines improve readability by setting off sections of the code that are logically related. A blank line should always be used in the following circumstances:

1. After the #include section.
2. After blocks of #defines of constants, and before and after #defines of macros.
3. Between structure declarations.
4. Between procedures.
5. After local variable declarations, and between the opening brace of a function and the first statement of the function if there are no local variable declarations.

10.2 Blank Spaces

Blank spaces should be used in following circumstances:

1. A **keyword**[15] and its opening parenthesis should be separated by a space. Blanks should not be used between a **procedure** name (or macro call) and its opening parenthesis. This helps to distinguish keywords from procedure calls.

---

15. Note that **sizeof** and **return** are keywords, whereas **strlen** and **exit** are not.

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

2. Blanks should appear after the commas in argument lists.
3. All binary operators except **.** and –> should be separated from their operands by blanks[16]. 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. For example,

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

4. The expressions in a **for** statement should be separated by blanks, e.g.,

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

5. 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);
```

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

11. Naming Conventions

Naming conventions 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 the code.

Here are some rules that should be followed:

1. 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, and **p** for pointers. Avoid variable **l** because it is hard to distinguish **l** from **1** on some printers and displays.
2. 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**.

---

16. 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.

3. Separate "words" in a long variable name with underscores, e.g., **create_panel_item**[17].
4. **#define** names for constants should be in all CAPS.
5. Two conventions are used for named types, i.e., **typedef**s. The name can end in "-t" or its first letter can be capitalized. For example,

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

OR

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

6. Macro names should be all CAPS except if the macro also exists as a function. In the latter case, the name should be only in lower case.
7. 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**, that can cause confusion.
8. The individual items of **enum**s should be unique. **enum** names can be guaranteed uniqueness by prefixing them with a tag identifying the package or module to which they belong. For example,

```
enum rainbow { rb_red, rb_orange, rb_green, rb_blue };
```

9. Every external name should be prefixed by a short unique name. This name should identify the module associated with the external name. For example,

```
int nfs_stat;     /* variable belongs in nfs module */
int vm_stat;      /* variable belongs to vm module */
```

12. Programming Practice

1. Numerical constants should not be coded directly[18].
2. 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**.
3. 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*.
4. Use of **goto** is discouraged. The main place where **goto**s can be employed is to break out of several levels of **switch**, **for**, and **while** nesting[19].

---

17. Mixed case names, e.g., **CreatePanelItem**, are strongly discouraged.

18. The constants -1, 0, and 1 may appear in a **for** loop as counter values.

19. 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.

5. Do not use a **goto** to branch to a label within a block. For example,

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

6. Do not bury a variable initialization in the middle of a long declaration. For example,

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

7. Avoid assigning several variables to the same value in a single statement. Do not use multiple assignments, because it is hard to read, for complex expressions. For Example,

```
        foo_bar.fb_name.fchar = bar_foo.fb_name.lchar = 'c';
```

8. It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator-precedence problems.

9. 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;
                ...
        };
#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
```

10. At the end of an **#ifdef** construct which is use to select among a required set of options (such as machine types), include a final **#else** clause. This clause should contain 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
      print unknown machine type;
#endif          /* machine type */
```

11. Do not 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++;
```

12. 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;
```

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

```
#define     BEGIN {
```

It makes the program unintelligible to all but the perpetrator.

13. Miscellaneous Comments

Try to make the structure of your program match the intent.

1. *Replace*

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

with

```
return (boolean_expression);
```

2. *Similarly,*

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

is usually clearer when written as:

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

3. 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[20]. 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, 1 for true, nothing else.
- is named so that the meaning of (say) a 'true' return is absolutely obvious. Call a predicate **is_valid** or **valid**, not **check_valid**.

4. Use of embedded assignments can improve the run-time performance. 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:

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

```
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. 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[21].

5. If an expression containing a binary operator appears before the **?**, it should be parenthesized:

```
(x >= 0) ? x : –x
```

6. Run *lint* to find obscure bugs.

14. 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. Individual projects may wish to establish additional standards beyond those given here to fit their needs.

---

21. 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.