Recommended C Style and Coding Standards

L.W. Cannon
R.A. Elliott
L.W. Kirchhoff
J.H. Miller
J.M. Milner
R.W. Mitze
E.P. Schan
N.O. Whittington

Bell Labs

Henry Spencer

Zoology Computer Systems
University of Toronto

David Keppel

EECS, UC Berkeley
CS&E, University of Washington

Mark Brader

SoftQuad Incorporated
Toronto

ABSTRACT

This document is an updated version of the *Indian Hill C Style and Coding Standards* paper, with modifications by the last three authors. It describes a recommended coding standard for C programs. The scope is coding style, not functional organization.

October 20, 1994

Recommended C Style and Coding Standards

L.W. Cannon
R.A. Elliott
L.W. Kirchhoff
J.H. Miller
J.M. Milner
R.W. Mitze
E.P. Schan
N.O. Whittington

Bell Labs

Henry Spencer

Zoology Computer Systems
University of Toronto

David Keppel

EECS, UC Berkeley
CS&E, University of Washington

Mark Brader

SoftQuad Incorporated
Toronto

1. Introduction

This document is a modified version of a document from a committee formed at AT&T's Indian Hill labs to establish a common set of coding standards and recommendations for the Indian Hill community. The scope of this work is C coding style. Good style should encourage consistent layout, improve portability, and reduce errors. This work does not cover functional organization, or general issues such as the use of *goto*s. We[1] have tried to combine previous work [1,6,8] on C style into a uniform set of standards that should be appropriate for any project using C, although parts are biased towards particular systems. Of necessity, these standards cannot cover all situations. Experience and informed judgement count for much. Programmers who encounter unusual situations should consult either experienced C programmers or code written by experienced C programmers (preferably following these rules).

The standards in this document are not of themselves required, but individual institutions or groups may adopt part or all of them as a part of program acceptance. It is therefore likely that others at your institution will code in a similar style. Ultimately, the goal of these standards is to increase portability, reduce maintenance, and above all improve clarity.

Many of the style choices here are somewhat arbitrary. Mixed coding style is harder to maintain than bad coding style. When changing existing code it is better to conform to the style (indentation, spacing, commenting, naming conventions) of the existing code than it is to blindly follow this document.

---

1. The opinions in this document do not reflect the opinions of all authors. This is still an evolving document. Please send comments and suggestions to pardo@cs.washington.edu or {rutgers,cornell,ucsd,ubc-cs,tektronix}!uw-beaver!june!pardo

''*To be clear is professional; not to be clear is unprofessional.*'' — Sir Ernest Gowers.

2. File Organization

A file consists of various sections that should be separated by several blank lines. Although there is no maximum length limit for source files, files with more than about 1000 lines are cumbersome to deal with. The editor may not have enough temp space to edit the file, compilations will go more slowly, etc. Many rows of asterisks, for example, present little information compared to the time it takes to scroll past, and are discouraged. Lines longer than 79 columns are not handled well by all terminals and should be avoided if possible. Excessively long lines which result from deep indenting are often a symptom of poorly-organized code.

2.1. File Naming Conventions

File names are made up of a base name, and an optional period and suffix. The first character of the name should be a letter and all characters (except the period) should be lower-case letters and numbers. The base name should be eight or fewer characters and the suffix should be three or fewer characters (four, if you include the period). These rules apply to both program files and default files used and produced by the program (e.g., ''rogue.sav'').

Some compilers and tools require certain suffix conventions for names of files [5]. The following suffixes are required:

- C source file names must end in *.c*
- Assembler source file names must end in *.s*

The following conventions are universally followed:

- Relocatable object file names end in *.o*
- Include header file names end in *.h*. An alternate convention that may be preferable in multi-language environments is to suffix both the language type and *.h* (e.g. ''foo.c.h'' or ''foo.ch'').
- Yacc source file names end in *.y*
- Lex source file names end in *.l*

C++ has compiler-dependent suffix conventions, including *.c*, *..c*, *.cc*, *.c.c*, and *.C*. Since much C code is also C++ code, there is no clear solution here.

In addition, it is conventional to use ''Makefile'' (not ''makefile'') for the control file for *make* (for systems that support it) and ''README'' for a summary of the contents of the directory or directory tree.

2.2. Program Files

The suggested order of sections for a program file is as follows:

1. First in the file is a prologue that tells what is in that file. 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 a list of the object names. The prologue may optionally contain author(s), revision control information, references, etc.
2. Any header file includes should be next. If the include is for a non-obvious reason, the reason should be commented. In most cases, system include files like *stdio.h* should be included before user include files.
3. Any defines and typedefs that apply to the file as a whole are next. One normal order is to have ''constant'' macros first, then ''function'' macros, then typedefs and enums.
4. Next come the global (external) data declarations, usually in the order: externs, non-static globals, static globals. 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 or embedded in structure declarations, indented to put the defines one level deeper than the first keyword of the declaration to which they apply.

5. The functions come last, and should be in some sort of meaningful order. Like functions should appear together. A ''breadth-first'' approach (functions on a similar level of abstraction together) is preferred over depth-first (functions defined as soon as possible before or after their calls). Considerable judgement is called for here. If defining large numbers of essentially-independent utility functions, consider alphabetical order.

2.3. Header Files

Header files are files that are included in other files prior to compilation by the C preprocessor. Some, such as *stdio.h*, are defined at the system level and must included by any program using the standard I/O library. Header files are also used to contain data declarations and defines that are needed by more than one program. Header files should be functionally organized, i.e., declarations for separate 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.

Avoid private header filenames that are the same as library header filenames. The statement #include "math.h" will include the standard library math header file if the intended one is not found in the current directory. If this is what you *want* to happen, comment this fact. Don't use absolute pathnames for header files. Use the <*name*> construction for getting them from a standard place, or define them relative to the current directory. The ''include-path'' option of the C compiler (–I on many systems) is the best way to handle extensive private libraries of header files; it permits reorganizing the directory structure without having to alter source files.

Header files that declare functions or external variables should be included in the file that defines the function or variable. That way, the compiler can do type checking and the external declaration will always agree with the definition.

Defining variables in a header file is often a poor idea. Frequently it is a symptom of poor partitioning of code between files. Also, some objects like typedefs and initialized data definitions cannot be seen twice by the compiler in one compilation. On some systems, repeating uninitialized declarations without the *extern* keyword also causes problems. Repeated declarations can happen if include files are nested and will cause the compilation to fail.

Header files should not be nested. The prologue for a header file should, therefore, describe what other headers need to be #included for the header to be functional. In extreme cases, where a large number of header files are to be included in several different source files, it is acceptable to put all common #includes in one include file.

It is common to put the following into each *.h* file to prevent accidental double-inclusion.

```
#ifndef EXAMPLE_H
#define EXAMPLE_H
...        /* body of example.h file */
#endif /* EXAMPLE_H */
```

This double-inclusion mechanism should not be relied upon, particularly to perform nested includes.

2.4. Other Files

It is conventional to have a file called ''README'' to document both ''the bigger picture'' and issues for the program as a whole. For example, it is common to include a list of all conditional compilation flags and what they mean. It is also common to list files that are machine dependent, etc.

3. Comments

''*When the code and the comments disagree, both are probably wrong.*'' — Norm Schryer

The comments should describe *what* is happening, *how* it is being done, what parameters mean, which globals are used and which are modified, and any restrictions or bugs. Avoid, however, comments that are clear from the code, as such information rapidly gets out of date. Comments that disagree with the code are of negative value. Short comments should be *what* comments, such as ''compute mean value'', rather than *how* comments such as ''sum of values divided by n''. C is not assembler; putting a comment at the top of a

3–10 line section telling what it does overall is often more useful than a comment on each line describing micrologic.

Comments should justify offensive code. The justification should be that something bad will happen if unoffensive code is used. Just making code faster is not enough to rationalize a hack; the performance must be _shown_ to be unacceptable without the hack. The comment should explain the unacceptable behavior and describe why the hack is a ‘‘good’’ fix.

Comments that describe data structures, algorithms, etc., should be in block comment form with the opening /* in columns 1-2, a * in column 2 before each line of comment text, and the closing */ in columns 2-3. An alternative is to have ** in columns 1-2, and put the closing */ also in 1-2.

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

```
/*
** Alternate format for block comments
*/
```

Note that _grep_ ’^.*’ will catch all block comments in the file[2]. Very long block comments such as drawn-out discussions and copyright notices often start with /* _in columns 1-2, no leading_ * before lines of text, and the closing */ _in columns 1-2. Block comments inside a function are appropriate, and they should be tabbed over to the same tab setting as the code that they describe. One-line comments alone on a line should be indented to the tab setting of the code that follows._

```
	if (argc > 1) {
		/* Get input file from command line. */
		if (freopen(argv[1], "r", stdin) == NULL) {
			perror (argv[1]);
		}
	}
```

Very short comments may appear on the same line as the code they describe, and should be tabbed over 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 == EXCEPTION) {
		b = TRUE;				/* special case */
	} else {
		b = isprime(a);			/* works only for odd a */
	}
```

4. Declarations

Global declarations should begin in column 1. All external data declaration should be preceded by the _extern_ keyword. If an external variable is an array that is defined with an explicit size, then the array bounds must be repeated in the extern declaration unless the size is always encoded in the array (e.g., a read-only character array that is always null-terminated). Repeated size declarations are particularly beneficial to someone picking up code written by another.

---

2. Some automated program-analysis packages use different characters before comment lines as a marker for lines with specific items of information. In particular, a line with a ‘–’ in a comment preceding a function is sometimes assumed to be a one-line summary of the function’s purpose.

The ‘‘pointer’’ qualifier, ‘*’, should be with the variable name rather than with the type.

```
char		*s, *t, *u;
```

instead of

```
char*	s, t, u;
```

which is wrong, since ‘t’ and ‘u’ do not get declared as pointers.

Unrelated declarations, even of the same type, should be on separate lines. A comment describing the role of the object being declared should be included, with the exception that a list of #*define*d constants do not need comments if the constant names are sufficient documentation. The names, values, and comments are usually tabbed so that they line up underneath each other. Use the tab character rather than blanks (spaces). 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 in column 1.

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

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

These defines are sometimes put right after the declaration of type, within the struct declaration, with enough tabs after the ‘#’ to indent define one level more than the structure member declarations. When the actual values are unimportant, the *enum* facility is better[3].

```
enum bt { KETCH=1, YAWL, SLOOP, SQRIG, MOTOR };
struct boat {
	int		wllength;	/* water line length in meters */
	enum bt	type;		/* what kind of boat */
	long		sailarea;	/* sail area in square mm */
};
```

Any variable whose initial value is important should be *explicitly* initialized, or at the very least should be commented to indicate that C’s default initialization to zero is being relied upon. The empty initializer, ‘‘{}’’*, should never be used. Structure initializations should be fully parenthesized with braces. Constants used to initialize longs should be explicitly long. Use capital letters; for example two long ‘‘2l*’’ looks a lot like ‘‘21’’*, the number twenty-one.*

```
int		x = 1;
char		*msg = "message";
struct boat	winner[] = {
	{ 40, YAWL, 6000000L },
	{ 28, MOTOR, 0L },
	{ 0 },
};
```

---

3. *enum*s might be better anyway.

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 usage should be commented to make it clear that another file's variables are being used; the comment should name the other file. If your debugger hides static objects you need to see during debugging, declare them as STATIC *and #define STATIC* as needed.

The most important types should be highlighted by typedeffing them, even if they are only integers, as the unique name makes the program easier to read (as long as there are only a *few* things typedeffed to integers!). Structures may be typedeffed when they are declared. Give the struct and the typedef the same name.

```
typedef struct splodge_t {
	int	sp_count;
	char	*sp_name, *sp_alias;
} splodge_t;
```

The return type of functions should always be declared. If function prototypes are available, use them. One common mistake is to omit the declaration of external math functions that return *double*. The compiler then assumes that the return value is an integer and the bits are dutifully converted into a (meaningless) floating point value.

''*C takes the point of view that the programmer is always right.*'' — Michael DeCorte

5. Function Declarations

Each function should be preceded by a block comment prologue that gives a short description of what the function does and (if not clear) how to use it. Discussion of non-trivial design decisions and side-effects is also appropriate. Avoid duplicating information clear from the code.

The function return type should be alone on a line, (optionally) indented one stop[4]. Do not default to *int*; if the function does not return a value then it should be given return type *void*[5]. If the value returned requires a long explanation, it should be given in the prologue; otherwise it can be on the same line as the return type, tabbed over. The function name (and the formal parameter list) should be alone on a line, in column 1. Destination (return value) parameters should generally be first (on the left). All formal parameter declarations, local declarations and code within the function body should be tabbed over one stop. The opening brace of the function body should be alone on a line beginning in column 1.

Each parameter should be declared (do not default to *int*). In general the role of each variable in the function should be described. This may either be done in the function comment or, if each declaration is on its own line, in a comment on that line. Loop counters called ''i'', string pointers called ''s'', and integral types called ''c'' and used for characters are typically excluded. If a group of functions all have a like parameter or local variable, it helps to call the repeated variable by the same name in all functions. (Conversely, avoid using the same name for different purposes in related functions.) Like parameters should also appear in the same place in the various argument lists.

Comments for parameters and local variables should be tabbed so that they line up underneath each other. Local variable declarations should be separated from the function's statements by a blank line.

Be careful when you use or declare functions that take a variable number of arguments (''varargs''). There is no truly portable way to do varargs in C. Better to design an interface that uses a fixed number of arguments. If you must have varargs, use the library macros for declaring functions with variant argument lists.

---

4. ''Tabstops'' can be blanks (spaces) inserted by your editor in clumps of 2, 4, or 8. Use actual tabs where possible.

5. #define void or #define void int for compilers without the *void* keyword.

If the function uses any external variables (or functions) that are not declared globally in the file, these should have their own declarations in the function body using the *extern* keyword.

Avoid local declarations that override declarations at higher levels. In particular, local variables should not be redeclared in nested blocks. Although this is valid C, the potential confusion is enough that *lint* will complain about it when given the –h option.

6. Whitespace

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

— Dishonorable mention, Obfuscated C Code Contest, 1984.
Author requested anonymity.

Use vertical and horizontal whitespace generously. Indentation and spacing should reflect the block structure of the code; e.g., there should be at least 2 blank lines between the end of one function and the comments for the next.

A long string of conditional operators should be split onto separate lines.

```
if (foo->next==NULL && totalcount<needed && needed<=MAX_ALLOT
        && server_active(current_input)) { ...
```

Might be better as

```
if (foo->next == NULL
        && totalcount < needed && needed <= MAX_ALLOT
        && server_active(current_input))
{
        ...
```

Similarly, elaborate *for* loops should be split onto different lines.

```
for (curr = *listp, trail = listp;
        curr != NULL;
        trail = &(curr->next), curr = curr->next )
{
        ...
```

Other complex expressions, particularly those using the ternary *?:* operator, are best split on to several lines, too.

```
c = (a == b)
        ? d + f(a)
        : f(b) - d;
```

Keywords that are followed by expressions in parentheses should be separated from the left parenthesis by a blank. (The *sizeof* operator is an exception.) Blanks should also appear after commas in argument lists to help separate the arguments visually. On the other hand, macro definitions with arguments must not have a blank between the name and the left parenthesis, otherwise the C preprocessor will not recognize the argument list.

7. Examples

```
/*
 *      Determine if the sky is blue by checking that it isn't night.
 *      CAVEAT: Only sometimes right.  May return TRUE when the answer
 *      is FALSE.  Consider clouds, eclipses, short days.
 *      NOTE: Uses 'hour' from 'hightime.c'.  Returns 'int' for
 *      compatibility with the old version.
 */
        int                                     /* true or false */
skyblue()
{
        extern int      hour;           /* current hour of the day */

        return (hour >= MORNING && hour <= EVENING);
}

/*
 *      Find the last element in the linked list
 *      pointed to by nodep and return a pointer to it.
 *      Return NULL if there is no last element.
 */
        node_t *
tail(nodep)
        node_t  *nodep;                 /* pointer to head of list */
{
        register node_t *np;            /* advances to NULL */
        register node_t *lp;            /* follows one behind np */

        if (nodep == NULL)
                return (NULL);
        for (np = lp = nodep; np != NULL; lp = np, np = np->next)
                ;       /* VOID */
        return (lp);
}
```

8. Simple Statements

There should be only one statement per line unless the statements are very closely related.

```
case FOO:       oogle (zork);  boogle (zork);  break;
case BAR:       oogle (bork);  boogle (zork);  break;
case BAZ:       oogle (gork);  boogle (bork);  break;
```

The null body of a *for* or *while* loop should be alone on a line and commented so that it is clear that the null body is intentional and not missing code.

```
while (*dest++ = *src++)
        ;       /* VOID */
```

Do not default the test for non-zero, i.e.

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

is better than

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

even though FAIL may have the value 0 which C considers to be false. An explicit test will help you out later when somebody decides that a failure return should be –1 instead of 0. Explicit comparison should be used even if the comparison value will never change; e.g., ‘‘if (!(bufsize % sizeof(int)))’’ should be written instead as ‘‘if ((bufsize % sizeof(int)) == 0)’’ to reflect the *numeric* (not *boolean*) nature of the test. A frequent trouble spot is using strcmp *to test for string equality, where the result should never ever be defaulted. The preferred approach is to define a macro STREQ.*

```
#define STREQ(a, b) (strcmp((a), (b)) == 0)
```

The non-zero test *is* often defaulted for predicates and other functions or expressions which meet the following restrictions:

- Evaluates to 0 for false, nothing else.
- Is named so that the meaning of (say) a ‘true’ return is absolutely obvious. Call a predicate *isvalid* or *valid,* not *checkvalid.*

It is common practice to declare a boolean type ‘‘bool’’ in a global include file. The special names improve readability immensely.

```
typedef int         bool;
#define FALSE   0
#define TRUE    1
```

or

```
typedef enum { NO=0, YES } bool;
```

Even with these declarations, do not check a boolean value for equality with 1 (TRUE, YES, etc.); instead test for inequality with 0 (FALSE, NO, etc.). Most functions are guaranteed to return 0 if false, but only non-zero if true. Thus,

```
if (func() == TRUE) { ...
```

must be written

```
if (func() != FALSE) { ...
```

It is even better (where possible) to rename the function/variable or rewrite the expression so that the meaning is obvious without a comparison to true or false (e.g., rename to *isvalid()*).

There is a time and a place for embedded assignment statements. In some constructs there is no better way to accomplish the results without making the code bulkier and less readable.

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

The ++ and −− operators count as assignment statements. So, for many purposes, do functions with side effects. Using embedded assignment statements 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,

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

should not be replaced by

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

even though the latter may save one cycle. 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.

Goto statements should be used sparingly, as in any well-structured code. The main place where they can be usefully employed is to break out of several levels of *switch*, *for*, and *while* nesting, although 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.

```
		for (...) {
			while (...) {
				...
				if (disaster)
					goto error;

			}
		}
		...
	error:
		clean up the mess
```

When a *goto* is necessary the accompanying label should be alone on a line and tabbed one stop to the left of the code that follows. The goto should be commented (possibly in the block header) as to its utility and purpose. *Continue* should be used sparingly and near the top of the loop. *Break* is less troublesome.

Parameters to non-prototyped functions sometimes need to be promoted explicitly. If, for example, a function expects a 32-bit *long* and gets handed a 16-bit *int* instead, the stack can get misaligned. Problems occur with pointer, integral, and floating-point values.

9. Compound Statements

A compound statement is a list of statements enclosed by braces. There are many common ways of formatting the braces. Be consistent with your local standard, if you have one, or pick one and use it consistently. When editing someone else's code, *always* use the style used in that code.

```
	control {
		statement;
		statement;
	}
```

The style above is called ‘‘K&R style’’, and is preferred if you haven't already got a favorite. With K&R style, the else part of an *if-else* statement and the while *part of a do-while statement should appear on the same line as the close brace. With most other styles, the braces are always alone on a line.*

When a block of code has several labels (unless there are a lot of them), the labels are placed on separate lines. The fall-through feature of the C *switch* statement, (that is, when there is no break *between a code segment and the next case* statement) must be commented for future maintenance. A lint-style comment/directive is best.

```
	switch (expr) {
	case ABC:
	case DEF:
		statement;
		break;
	case UVW:
		statement;
		/*FALLTHROUGH*/
	case XYZ:
		statement;
		break;
	}
```

Here, the last break is unnecessary, but is required because it prevents a fall-through error if another case is added later after the last one. The default case, if used, should be last and does not require a break if it is last.

Whenever an *if-else* statement has a compound statement for either the *if* or *else* section, the statements of both the *if* and *else* sections should both be enclosed in braces (called *fully bracketed syntax*).

```
if (expr) {
        statement;
} else {
        statement;
        statement;
}
```

Braces are also essential in *if-if-else* sequences with no second *else* such as the following, which will be parsed incorrectly if the brace after (ex1) *and its mate are omitted:*

```
if (ex1) {
        if (ex2) {
                funca();
        }
} else {
        funcb();
}
```

An *if-else* with *else if* should be written with the *else* conditions left-justified.

```
if (STREQ (reply, "yes")) {
        statements for yes
        ...
} else if (STREQ (reply, "no")) {
        ...
} else if (STREQ (reply, "maybe")) {
        ...
} else {
        statements for default
        ...
}
```

The format then looks like a generalized *switch* statement and the tabbing reflects the switch between exactly one of several alternatives rather than a nesting of statements.

Do-while loops should always have braces around the body.

The following code is very dangerous:

```
#ifdef CIRCUIT
#       define CLOSE_CIRCUIT(circno)   { close_circ(circno); }
#else
#       define CLOSE_CIRCUIT(circno)
#endif

        ...
        if (expr)
                statement;
        else
                CLOSE_CIRCUIT(x)
        ++i;
```

Note that on systems where CIRCUIT is not defined the statement ‘‘++i;’’ will only get executed when expr is false! This example points out both the value of naming macros with CAPS and of making code fully-bracketed.

Sometimes an *if* causes an unconditional control transfer via break, *continue*, goto, *or return*. The *else* should be implicit and the code should not be indented.

```
if (level > limit)
        return (OVERFLOW)
normal();
return (level);
```

The ‘‘flattened’’ indentation tells the reader that the boolean test is invariant over the rest of the enclosing block.

10. Operators

Unary operators should not be separated from their single operand. Generally, all binary operators except ‘**.’ and ‘**–>’ should be separated from their operands by blanks. Some judgement 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.

If you think an expression will be hard to read, consider breaking it across lines. Splitting at the lowest-precedence operator near the break is best. Since C has some unexpected precedence rules, expressions involving mixed operators should be parenthesized. Too many parentheses, however, can make a line *harder* to read because humans aren’t good at parenthesis-matching.

There is a time and place for the binary comma operator, but generally it should be avoided. The comma operator is most useful to provide multiple initializations or operations, as in *for* statements. Complex expressions, for instance those with nested ternary *?:* operators, can be confusing and should be avoided if possible. There are some macros like getchar *where both the ternary operator and comma operators are useful. The logical expression operand before the ?: should be parenthesized and both return values must be the same type.*

11. Naming Conventions

Individual projects will no doubt have their own naming conventions. There are some general rules however.

- Names with leading and trailing underscores are reserved for system purposes and should not be used for any user-created names. Most systems use them for names that the user should not have to know. If you must have your own private identifiers, begin them with a letter or two identifying the package to which they belong.
- #define constants should be in all CAPS.
- Enum constants are Capitalized or in all CAPS
- Function, typedef, and variable names, as well as struct, union, and enum tag names should be in lower case.
- Many macro ‘‘functions’’ are in all CAPS. Some macros (such as getchar and putchar) are in lower case since they may also exist as functions. Lower-case macro names are only acceptable if the macros behave like a function call, that is, they evaluate their parameters *exactly* once and do not assign values to named parameters. Sometimes it is impossible to write a macro that behaves like a function even though the arguments are evaluated exactly once.
- Avoid names that differ only in case, like *foo* and *Foo*. Similarly, avoid *foobar* and *foo_bar*. The potential for confusion is considerable.
- Similarly, avoid names that look like each other. On many terminals and printers, ‘l’, ‘1’ and ‘I’ look quite similar. A variable named ‘l’ is particularly bad because it looks so much like the constant ‘1’.

In general, global names (including *enum*s) should have a common prefix identifying the module that they belong with. Globals may alternatively be grouped in a global structure. Typedeffed names often have ‘‘_t’’ *appended to their name.*

Avoid names that might conflict with various standard library names. Some systems will include more library code than you want. Also, your program may be extended someday.

12. Constants

Numerical constants should not be coded directly. The *#define* feature of the C preprocessor should be used to give constants meaningful names. Symbolic constants make the code easier to read. Defining the value in one place also makes it easier to administer large programs since the constant value can be changed uniformly by changing only the define. The enumeration data type is a better way to declare variables that take on only a discrete set of values, since additional type checking is often available. At the very least, any directly-coded numerical constant must have a comment explaining the derivation of the value.

Constants should be defined consistently with their use; e.g. use 540.0 for a float instead of 540 with an implicit float cast. There are some cases where the constants 0 and 1 may appear as themselves instead of as defines. For example if a *for* loop indexes through an array, then

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

is reasonable while the code

```
door_t *front_door = opens(door[i], 7);
if (front_door == 0)
        error("can't open %s\n", door[i]);
```

is not. In the last example front_door *is a pointer. When a value is a pointer it should be compared to NULL* instead of 0. *NULL* is available either as part of the standard I/O library's header file *stdio.h* or in *stdlib.h* for newer systems. Even simple values like 1 or 0 are often better expressed using defines like *TRUE* and *FALSE* (sometimes *YES* and *NO* read better).

Simple character constants should be defined as character literals rather than numbers. Non-text characters are discouraged as non-portable. If non-text characters are necessary, particularly if they are used in strings, they should be written using a escape character of three octal digits rather than one (e.g., '\007'). Even so, such usage should be considered machine-dependent and treated as such.

13. Macros

Complex expressions can be used as macro parameters, and operator-precedence problems can arise unless all occurrences of parameters have parentheses around them. There is little that can be done about the problems caused by side effects in parameters except to avoid side effects in expressions (a good idea anyway) and, when possible, to write macros that evaluate their parameters exactly once. There are times when it is impossible to write macros that act exactly like functions.

Some macros also exist as functions (e.g., getc and fgetc). The macro should be used in implementing the function so that changes to the macro will be automatically reflected in the function. Care is needed when interchanging macros and functions since function parameters are passed by value, while macro parameters are passed by name substitution. Carefree use of macros requires that they be declared carefully.

Macros should avoid using globals, since the global name may be hidden by a local declaration. Macros that change named parameters (rather than the storage they point at) or may be used as the left-hand side of an assignment should mention this in their comments. Macros that take no parameters but reference variables, are long, or are aliases for function calls should be given an empty parameter list, e.g.,

```
#define  OFF_A()        (a_global+OFFSET)
#define  BORK()         (zork())
#define  SP3()   if (b) { int x; av = f (&x); bv += x; }
```

Macros save function call/return overhead, but when a macro gets long, the effect of the call/return becomes negligible, so a function should be used instead.

In some cases it is appropriate to make the compiler insure that a macro is terminated with a semicolon.

```
if (x==3)
    SP3( );
else
    BORK( );
```

If the semicolon is omitted after the call to SP3, then the else will (silently!) become associated with the if in the SP3 macro. With the semicolon, the else doesn't match **any** if**! The macro SP3** can be written safely as

```
#define SP3( ) \
        do { if (b) { int x; av = f (&x); bv += x; }} while (0)
```

Writing out the enclosing *do-while* by hand is awkward and some compilers and tools may complain that there is a constant in the ''while'' *conditional. A macro for declaring statements may make programming easier.*

```
#ifdef lint
        static int ZERO;
#else
#       define ZERO 0
#endif
#define STMT( stuff )           do { stuff } while (ZERO)
```

Declare SP3 with

```
#define SP3( ) \
        STMT( if (b) { int x; av = f (&x); bv += x; } )
```

Using STMT *will help prevent small typos from silently changing programs.*

Except for type casts, sizeof*, and hacks such as the above, macros should contain keywords only if the entire macro is surrounded by braces.*

14. Conditional Compilation.

Conditional compilation is useful for things like machine-dependencies, debugging, and for setting certain options at compile-time. Beware of conditional compilation. Various controls can easily combine in unforeseen ways. If you #ifdef machine dependencies, make sure that when no machine is specified, the result is an error, not a default machine. (Use ''#error**'' and indent it so it works with older compilers.) If you #ifdef optimizations, the default should be the unoptimized code rather than an uncompilable program. Be sure to test the unoptimized code.**

Note that the text inside of an #ifdeffed section may be scanned (processed) by the compiler, even if the #ifdef is false. Thus, even if the #ifdeffed part of the file never gets compiled (e.g., #ifdef COMMENT**), it cannot be arbitrary text.**

Put #ifdefs in header files instead of source files when possible. Use the #ifdefs to define macros that can be used uniformly in the code. For instance, a header file for checking memory allocation might look like (omitting definitions for REALLOC **and FREE**):

```
#ifdef DEBUG
        extern void *mm_malloc();
#       define MALLOC(size) (mm_malloc(size))
#else
        extern void *malloc();
#       define MALLOC(size) (malloc(size))
#endif
```

Conditional compilation should generally be on a feature-by-feature basis. Machine or operating system dependencies should be avoided in most cases.

```
#ifdef BSD4
        long t = time ((long *)NULL);
#endif
```

The preceding code is poor for two reasons: there may be 4BSD systems for which there is a better choice, and there may be non-4BSD systems for which the above *is* the best code. Instead, use *define* symbols such as TIME_LONG *and TIME_STRUCT* and define the appropriate one in a configuration file such as *config.h*.

15. Debugging

''*C Code. C code run. Run, code, run... PLEASE!!!*'' — Barbara Tongue

If you use *enum*s, the first enum constant should have a non-zero value, or the first constant should indicate an error.

```
enum { STATE_ERR, STATE_START, STATE_NORMAL, STATE_END } state_t;
enum { VAL_NEW=1, VAL_NORMAL, VAL_DYING, VAL_DEAD } value_t;
```

Uninitialized values will then often ''catch themselves''.

Check for error return values, even from functions that ''can't'' fail. Consider that close() and fclose() can and do fail, even when all prior file operations have succeeded. Write your own functions so that they test for errors and return error values or abort the program in a well-defined way. Include a lot of debugging and error-checking code and leave most of it in the finished product. Check even for ''impossible'' errors. [8]

Use the *assert* facility to insist that each function is being passed well-defined values, and that intermediate results are well-formed.

Build in the debug code using as few #ifdefs as possible. For instance, if ''mm_malloc'' is a debugging memory allocator, then MALLOC will select the appropriate allocator, avoids littering the code with #ifdefs, and makes clear the difference between allocation calls being debugged and extra memory that is allocated only during debugging.

```
#ifdef DEBUG
#       define MALLOC(size)  (mm_malloc(size))
#else
#       define MALLOC(size)  (malloc(size))
#endif
```

Check bounds even on things that ''can't'' overflow. A function that writes on to variable-sized storage should take an argument maxsize that is the size of the destination. If there are times when the size of the destination is unknown, some 'magic' value of maxsize should mean ''no bounds checks''. When bound checks fail, make sure that the function does something useful such as abort or return an error status.

```
/*
 * INPUT: A null-terminated source string 'src' to copy from and
 * a 'dest' string to copy to.  'maxsize' is the size of 'dest'
 * or UINT_MAX if the size is not known.  'src' and 'dest' must
 * both be shorter than UINT_MAX, and 'src' must be no longer than
 * 'dest'.
 * OUTPUT: The address of 'dest' or NULL if the copy fails.
 * 'dest' is modified even when the copy fails.
 */
	char *
copy (dest, maxsize, src)
	char *dest, *src;
	unsigned maxsize;
{
	char *dp = dest;

	while (maxsize-- > 0)
		if ((*dp++ = *src++) == '\0')
			return (dest);

	return (NULL);
}
```

In all, remember that a program that produces wrong answers twice as fast is infinitely slower. The same is true of programs that crash occasionally or clobber valid data.

16. Portability

''C combines the power of assembler with the portability of assembler.''
— Anonymous, alluding to Bill Thacker.

The advantages of portable code are well known. This section gives some guidelines for writing portable code. Here, ''portable'' means that a source file can be compiled and executed on different machines with the only change being the inclusion of possibly different header files and the use of different compiler flags. The header files will contain #defines and typedefs that may vary from machine to machine. In general, a new ''machine'' is different hardware, a different operating system, a different compiler, or any combination of these. Reference [1] contains useful information on both style and portability. The following is a list of pitfalls to be avoided and recommendations to be considered when designing portable code:

- Write portable code first, worry about detail optimizations only on machines where they prove necessary. Optimized code is often obscure. Optimizations for one machine may produce worse code on another. Document performance hacks and localize them as much as possible. Documentation should explain *how* it works and *why* it was needed (e.g., ''loop executes 6 zillion times'').
- 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 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.
- Organize source files 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. Comment the machine dependence in the headers of the appropriate files.

- Any behavior that is described as ‘‘implementation defined’’ should be treated as a machine (compiler) dependency. Assume that the compiler or hardware does it some completely screwy way.
- Pay attention to word sizes. Objects may be non-intuitive sizes, Pointers are not always the same size as *int*s, the same size as each other, or freely interconvertible. The following table shows bit sizes for basic types in C for various machines and compilers.

type	pdp11 series	VAX/11	68000 family	Cray-2	Unisys 1100	Harris H800	80386
char	8	8	8	8	9	8	8
short	16	16	8/16	64(32)	18	24	8/16
int	16	32	16/32	64(32)	36	24	16/32
long	32	32	32	64	36	48	32
char*	16	32	32	64	72	24	16/32/48
int*	16	32	32	64(24)	72	24	16/32/48
int(*)( )	16	32	32	64	576	24	16/32/48

Some machines have more than one possible size for a given type. The size you get can depend both on the compiler and on various compile-time flags. The following table shows ‘‘safe’’ type sizes on the majority of systems. Unsigned numbers are the same bit size as signed numbers.

Type	Minimum # Bits	No Smaller Than
char	8	
short	16	char
int	16	short
long	32	int
float	24	
double	38	float
any *	14	
char *	15	any *
void *	15	any *

- The *void** type is guaranteed to have enough bits of precision to hold a pointer to any data object. The *void(*)( )* type is guaranteed to be able to hold a pointer to any function. Use these types when you need a generic pointer. (Use *char** and *char(*)( )*, respectively, in older compilers). Be sure to cast pointers back to the correct type before using them.
- Even when, say, an *int** and a *char** are the same *size*, they may have different *formats*. For example, the following will fail on some machines that have sizeof(int*) *equal to sizeof(char*)*. The code fails because free *expects a char* and gets passed an int**.

```
int *p = (int *) malloc (sizeof(int));
free (p);
```

- Note that the *size* of an object does not guarantee the *precision* of that object. The Cray-2 may use 64 bits to store an *int*, but a *long* cast into an *int* and back to a *long* may be truncated to 32 bits.
- The integer *constant* zero may be cast to any pointer type. The resulting pointer is called a *null pointer* for that type, and is different from any other pointer of that type. A null pointer always compares equal to the constant zero. A null pointer might *not* compare equal with a variable that has the value zero. Null pointers are *not* always stored with all bits zero. Null pointers for two different types are sometimes different. A null pointer of one type cast in to a pointer of another type will be cast in to the null pointer for that second type.

- On ANSI compilers, when two pointers of the same type access the same storage, they will compare as equal. When non-zero integer constants are cast to pointer types, they may become identical to other pointers. On non-ANSI compilers, pointers that access the same storage may compare as different. The following two pointers, for instance, may or may not compare equal, and they may or may not access the same storage[6].

  ```
  ((int *) 2 )
  ((int *) 3 )
  ```

  If you need 'magic' pointers other than NULL, either allocate some storage or treat the pointer as a machine dependence.

  ```
  extern int x_int_dummy;          /* in x.c */
  #define X_FAIL    (NULL)
  #define X_BUSY    (&x_int_dummy)

  #define X_FAIL    (NULL)
  #define X_BUSY    MD_PTR1       /* MD_PTR1 from "machdep.h" */
  ```

- Floating-point numbers have both a *precision* and a *range*. These are independent of the size of the object. Thus, overflow (underflow) for a 32-bit floating-point number will happen at different values on different machines. Also, 4.9 times 5.1 will yield two different numbers on two different machines. Differences in rounding and truncation can give surprisingly different answers.
- On some machines, a *double* may have *less* range or precision than a *float*.
- On some machines the first half of a *double* may be a *float* with similar value. Do *not* depend on this.
- Watch out for signed characters. On some VAXes, for instance, characters are sign extended when used in expressions, which is not the case on many other machines. Code that assumes signed/unsigned is unportable. For example, array[c] won't work if c is supposed to be positive and is instead signed and negative. If you must assume signed or unsigned characters, comment them as SIGNED or UNSIGNED. Unsigned behavior can be guaranteed with unsigned char.
- Avoid assuming ASCII. If you must assume, document and localize. Remember that characters may hold (much) more than 8 bits.
- Code that takes advantage of the two's complement representation of numbers on most machines should not be used. Optimizations that replace arithmetic operations with equivalent shifting operations are particularly suspect. If absolutely necessary, machine-dependent code should be #ifdeffed or operations should be performed by #ifdeffed macros. You should weigh the time savings with the potential for obscure and difficult bugs when your code is moved.
- In general, if the word size or value range is important, typedef ''sized'' types. Large programs should have a central header file which supplies typedefs for commonly-used width-sensitive types, to make it easier to change them and to aid in finding width-sensitive code. Unsigned types other than *unsigned int* are highly compiler-dependent. 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.
- Data *alignment* is also important. For instance, on various machines a 4-byte integer may start at any address, start only at an even address, or start only at a multiple-of-four address. Thus, a particular structure may have its elements at different offsets on different machines, even when given elements are the same size on all machines. Indeed, a structure of a 32-bit pointer and an 8-bit character may be 3 sizes on 3 different machines. As a corollary, pointers to objects may not be interchanged freely; saving an integer through a pointer to 4 bytes starting at an odd address will

---

6. The code may also fail to compile, fault on pointer creation, fault on pointer comparison, or fault on pointer dereferences.

sometimes work, sometimes cause a core dump, and sometimes fail silently (clobbering other data in the process). Pointer-to-character is a particular trouble spot on machines which do not address to the byte. 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.

- The bytes of a word are of increasing significance with increasing address on machines such as the VAX (little-endian) and of decreasing significance with increasing address on other machines such as the 68000 (big-endian). The order of bytes in a word and of words in larger objects (say, a double word) might not be the same. Hence any code that depends on the left-right orientation of bits in an object deserves special scrutiny. Bit fields within structure members will only be portable so long as two separate fields are never concatenated and treated as a unit. [1,3] Actually, it is nonportable to concatenate *any* two variables.
- There may be unused holes in structures. Suspect unions used for type cheating. Specifically, a value should not be stored as one type and retrieved as another. An explicit tag field for unions may be useful.
- Different compilers use different conventions for returning structures. This causes a problem when libraries return structure values to code compiled with a different compiler. Structure pointers are not a problem.
- Do not make assumptions about the parameter passing mechanism. especially pointer sizes and parameter evaluation order, size, etc. The following code, for instance, is *very* nonportable.

```
        c = foo (getchar(), getchar());

        char
foo (c1, c2, c3)
        char c1, c2, c3;
{
        char bar = *(&c1 + 1);
        return (bar);                   /* often won't return c2 */
}
```

This example has lots of problems. The stack may grow up or down (indeed, there need not even be a stack!). Parameters may be widened when they are passed, so a *char* might be passed as an *int*, for instance. Arguments may be pushed left-to-right, right-to-left, in arbitrary order, or passed in registers (not pushed at all). The order of evaluation may differ from the order in which they are pushed. One compiler may use several (incompatible) calling conventions.

- On some machines, the null character pointer ((char *)0) is treated the same way as a pointer to a null string. Do *not* depend on this.
- Do not modify string constants[7]. One particularly notorious (bad) example is

```
s = "/dev/tty??";
strcpy (&s[8], ttychars);
```

- The address space may have holes. Simply **computing** the address of an unallocated element in an array (before or after the actual storage of the array) may crash the program. If the address is used in a comparison, sometimes the program will run but clobber data, give wrong answers, or loop forever. In ANSI C, a pointer into an array of objects may legally point to the first element after the end of the array; this is usually safe in older implementations. This ''outside'' pointer may not be dereferenced.

---

7. Some libraries attempt to modify and then restore read-only string variables. Programs sometimes won't port because of these broken libraries. The libraries are getting better.

- Only the == and != comparisons are defined for all pointers of a given type. It is only portable to use <, <=, >, or >= to compare pointers when they both point in to (or to the first element after) the same array. It is likewise only portable to use arithmetic operators on pointers that both point into the same array or the first element afterwards.
- Word size also affects shifts and masks. The following code will clear only the three rightmost bits of an *int* on *some* 68000s. On other machines it will also clear the upper two bytes.

  ```
  x &= 0177770
  ```

  Use instead

  ```
  x &= ˜07
  ```

  which works properly on all machines. Bitfields do not have these problems.
- 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. Notorious examples include the following.

  ```
  a[i] = b[i++];
  ```

  In the above example, we know only that the subscript into b has not been incremented. The index into a could be the value of i either before or after the increment.

  ```
  struct bar_t { struct bar_t *next; } bar;
  bar->next = bar = tmp;
  ```

  In the second example, the address of ''bar->next'' may be computed before the value is assigned to ''bar''.

  ```
  bar = bar->next = tmp;
  ```

  In the third example, bar can be assigned before bar->next. Although this *appears* to violate the rule that ''assignment proceeds right-to-left'', it is a legal interpretation. Consider the following example:

  ```
  long i;
  short a[N];
  i = old
  i = a[i] = new;
  ```

  The value that ''i'' *is assigned must be a value that is typed as if assignment proceeded right-to-left. However, ''i''* may be assigned the value ''(long)(short)new'' *before ''a[i]''* is assigned to. Compilers do differ.
- Be suspicious of numeric values appearing in the code (''magic numbers'').
- Avoid preprocessor tricks. Tricks such as using /**/ for token pasting and macros that rely on argument string expansion will break reliably.

  ```
  #define FOO(string)(printf("string = %s",(string)))
  ...
  FOO(filename);
  ```

  Will only sometimes be expanded to

  ```
  (printf("filename = %s",(filename)))
  ```

  Be aware, however, that tricky preprocessors may cause macros to break *accidentally* on some machines. Consider the following two versions of a macro.

  ```
  #define LOOKUP(chr)        (a['c'+(chr)])/* Works as intended. */
  #define LOOKUP(c)          (a['c'+(c)])/* Sometimes breaks. */
  ```

  The second version of LOOKUP *can be expanded in two different ways and will cause code to* *break mysteriously.*

- Become familiar with existing library functions and defines. (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.) You should not be writing your own string compare routine, terminal control routines, or making your own defines for system structures. ''Rolling your own'' wastes your time and makes your code less readable, because another reader has to figure out whether you're doing something special in that reimplemented stuff to justify its existence. It also prevents your program from taking advantage of any microcode assists or other means of improving performance of system routines. Furthermore, it's a fruitful source of bugs. If possible, be aware of the *differences* between the common libraries (such as ANSI, POSIX, and so on).
- Use *lint* when it is available. It is a valuable tool for finding machine-dependent constructs as well as other inconsistencies or program bugs that pass the compiler. If your compiler has switches to turn on warnings, use them.
- Suspect labels inside blocks with the associated *switch* or *goto* outside the block.
- Wherever the type is in doubt, parameters should be cast to the appropriate type. Always cast NULL when it appears in non-prototyped function calls. Do not use function calls as a place to do type cheating. C has confusing promotion rules, so be careful. For example, if a function expects a 32-bit *long* and it is passed a 16-bit *int* the stack can get misaligned, the value can get promoted wrong, etc.
- Use explicit casts when doing arithmetic that mixes signed and unsigned values.
- The inter-procedural goto, *longjmp*, should be used with caution. Many implementations ''forget'' to restore values in registers. Declare critical values as *volatile* if you can or comment them as VOLATILE.
- Some linkers convert names to lower-case and some only recognize the first six letters as unique. Programs may break quietly on these systems.
- Beware of compiler extensions. If used, document and consider them as machine dependencies.
- A program cannot generally execute code in the data segment or write into the code segment. Even when it can, there is no guarantee that it can do so reliably.

17. ANSI C

Modern C compilers support some or all of the ANSI proposed standard C. Whenever possible, write code to run under standard C, and use features such as function prototypes, constant storage, and volatile storage. Standard C improves program performance by giving better information to optimizers. Standard C improves portability by insuring that all compilers accept the same input language and by providing mechanisms that try to hide machine dependencies or emit warnings about code that may be machine-dependent.

17.1. Compatibility

Write code that is easy to port to older compilers. For instance, conditionally #define new (standard) keywords such as *const* and *volatile* in a global *.h* file. Standard compilers pre-define the preprocessor symbol __STDC__[8]. *The void* type is hard to get right simply, since some older compilers understand void but not void*. It is easiest to create a new (machine- and compiler-dependent) VOIDP type, usually char* on older compilers.*

---

8. Some compilers predefine __STDC__ to be 0, in an attempt to indicate partial compliance with the ANSI C standard. Unfortunately, it is not possible to determine *which* ANSI facilities are provided. Thus, such compilers are broken. See the rule about ''don't write around a broken compiler unless you are forced to.''

```
#if __STDC__
        typedef void *voidp;
#       define COMPILER_SELECTED
#endif
#ifdef A_TARGET
#       define const
#       define volatile
#       define void int
        typedef char *voidp;
#       define COMPILER_SELECTED
#endif
#ifdef ...
        ...
#endif
#ifdef COMPILER_SELECTED
#       undef COMPILER_SELECTED
#else
        { NO TARGET SELECTED! }
#endif
```

Note that under ANSI C, the '#' for a preprocessor directive must be the first non-whitespace character on a line. Under older compilers it must be the first character on the line.

When a static function has a forward declaration, the forward declaration must include the storage class. For older compilers, the class must be ''*extern*''. For ANSI compilers, the class must be ''*static*''. but global functions must still be declared as ''*extern*''. Thus, forward declarations of static functions should use a #define such as FWD_STATIC *that is #ifdeffed as appropriate.*

An ''#ifdef NAME'' *should end with either* ''#endif'' or ''#endif /* NAME */'', *not with* ''#endif NAME''. *The comment should not be used on short #ifdefs, as it is clear from the code.*

ANSI *trigraphs* may cause programs with strings containing ''??'' *may break mysteriously*.

17.2. Formatting

The style for ANSI C is the same as for regular C, with two notable exceptions: storage qualifiers and parameter lists.

Because const and volatile have strange binding rules, each *const* or *volatile* object should have a separate declaration.

```
int const *s;                   /* YES */
int const *s, *t;       /* NO */
```

Prototyped functions merge parameter declaration and definition in to one list. Parameters should be commented in the function comment.

```
/*
 * 'bp': boat trying to get in.
 * 'stall': a list of stalls, never NULL.
 * returns stall number, 0 => no room.
 */
        int
enter_pier (boat_t const *bp, stall_t *stall)
{
        ...
```

17.3. Prototypes

Function prototypes should be used to make code more robust and to make it run faster. Unfortunately, the prototyped **declaration**

```
extern void bork (char c);
```

is incompatible with the **definition**

```
        void
bork (c)
        char c;
...
```

The prototype says that c **is to be passed as the most natural type for the machine, possibly a byte. The non-prototyped (backwards-compatible) definition implies that c** is always passed as an *int*[9]. If a function has promotable parameters then the caller and callee must be compiled identically. Either both must use function prototypes or neither can use prototypes. The problem can be avoided if parameters are promoted when the program is designed. For example, bork *can be defined to take an int parameter.*

The above declaration works if the definition is prototyped.

```
        void
bork (char c)
{
        ...
```

Unfortunately, the prototyped syntax will cause non-ANSI compilers to reject the program.

It *is* easy to write external declarations that work with both prototyping and with older compilers[10].

```
#if __STDC__
#       define PROTO(x) x
#else
#       define PROTO(x) ()
#endif

extern char **ncopies PROTO((char *s, short times));
```

Note that PROTO *must be used with double parentheses.*

In the end, it may be best to write in only one style (e.g., with prototypes). When a non-prototyped version is needed, it is generated using an automatic conversion tool.

17.4. Pragmas

Pragmas are used to introduce machine-dependent code in a controlled way. Obviously, pragmas should be treated as machine dependencies. Unfortunately, the syntax of ANSI pragmas makes it impossible to isolate them in machine-dependent headers.

Pragmas are of two classes. *Optimizations* may safely be ignored. Pragmas that change the system behavior (''required pragmas'') may not. Required pragmas should be #ifdeffed so that compilation will abort if no pragma is selected.

Two compilers may use a given pragma in two very different ways. For instance, one compiler may use ''haggis'' to signal an optimization. Another might use it to indicate that a given statement, if

---

9. Such automatic type promotion is called *widening*. For older compilers, the widening rules require that all *char* and *short* parameters are passed as *int*s and that *float* parameters are passed as *double*s.

10. Note that using PROTO violates the rule ''don't change the syntax via macro substitution.'' It is regrettable that there isn't a better solution.

reached, should terminate the program. Thus, when pragmas are used, they must always be enclosed in machine-dependent #ifdefs. Pragmas must always be #ifdefed out for non-ANSI compilers. Be sure to indent the '#' character on the #pragma, as older preprocessors will halt on it otherwise.

```
#if defined(__STDC__) && defined(USE_HAGGIS_PRAGMA)
	#pragma (HAGGIS)
#endif
```

> ''*The '#pragma' command is specified in the ANSI standard to have an arbitrary implementation-defined effect. In the GNU C preprocessor, '#pragma' first attempts to run the game 'rogue'; if that fails, it tries to run the game 'hack'; if that fails, it tries to run GNU Emacs displaying the Tower of Hanoi; if that fails, it reports a fatal error. In any case, preprocessing does not continue.*''
>
> — Manual for the GNU C preprocessor for GNU CC 1.34.

18. Special Considerations

This section contains some miscellaneous do's and don'ts.

- Don't change syntax via macro substitution. It makes the program unintelligible to all but the perpetrator.
- Don't use floating-point variables where discrete values are needed. Using a *float* for a loop counter is a great way to shoot yourself in the foot. Always test floating-point numbers as **<=** or **>=**, never use an exact comparison (**==** or **!=**).
- Compilers have bugs. Common trouble spots include structure assignment and bitfields. You cannot generally predict which bugs a compiler has. You *could* write a program that avoids all constructs that are known broken on all compilers. You won't be able to write anything useful, you might still encounter bugs, and the compiler might get fixed in the meanwhile. Thus, you should write ''around'' compiler bugs only when you are *forced* to use a particular buggy compiler.
- Do not rely on automatic beautifiers. The main person who benefits from good program style is the programmer him/herself, and especially in the early design of handwritten algorithms or pseudo-code. Automatic beautifiers can only be applied to complete, syntactically correct programs and hence are not available when the need for attention to white space and indentation is greatest. Programmers can do a better job of making clear the complete visual layout of a function or file, with the normal attention to detail of a careful programmer. (In other words, some of the visual layout is dictated by intent rather than syntax and beautifiers cannot read minds.) Sloppy programmers should learn to be careful programmers instead of relying on a beautifier to make their code readable.
- Accidental omission of the second ''='' of the logical compare is a problem. Use explicit tests. Avoid assignment with implicit test.

  ```
  abool = bbool;
  if (abool) { ...
  ```

  When embedded assignment *is* used, make the test explicit so that it doesn't get ''fixed'' later.

  ```
  while ((abool = bbool) != FALSE) { ...

  while (abool = bbool) { ...		/* VALUSED */

  while (abool = bbool, abool) { ...
  ```

- Explicitly comment variables that are changed out of the normal control flow, or other code that is likely to break during maintenance.

- Modern compilers will put variables in registers automatically. Use the *register* sparingly to indicate the variables that you think are most critical. In extreme cases, mark the 2-4 most critical values as register *and mark the rest as REGISTER.* The latter can be #defined to register *on those machines with many registers.*

19. Lint

Lint is a C program checker [2][11] that examines C source files to detect and report type incompatibilities, inconsistencies between function definitions and calls, potential program bugs, etc. The use of *lint* on all programs is strongly recommended, and it is expected that most projects will require programs to use *lint* as part of the official acceptance procedure.

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 during and after changes or additions to the code. *Lint* can find obscure bugs and insure portability before problems occur. Many messages from *lint* really do indicate something wrong. One fun story is about is about a program that was missing an argument to 'fprintf'.

```
fprintf ("Usage: foo -bar <file>\n");
```

The author never had a problem. But the program dumped core every time an ordinary user made a mistake on the command line. Many versions of lint will catch this.

Most options are worth learning. Some options may complain about legitimate things, but they will also pick up many botches. Note that –p[11] checks function-call type-consistency for only a subset of library routines, so programs should be linted both with and without –p for the best ''coverage''.

Lint also recognizes several special comments in the code. These comments both shut up *lint* when the code otherwise makes it complain, and also document special code.

20. Make

One other very useful tool is *make* [7]. During development, *make* recompiles only those modules that have been changed since the last time *make* was used. It can be used to automate other tasks, as well. Some common conventions include:

all	*always* makes all binaries
clean	remove all intermediate files
debug	make a test binary 'a.out' or 'debug'
depend	make transitive dependencies
install	install binaries, libraries, etc.
deinstall	back out of ''install''
mkcat	install the manual page(s)
lint	run lint
print/list	make a hard copy of all source files
shar	make a shar of all source files
spotless	make clean, use revision control to put away sources.
	Note: doesn't remove Makefile, although it is a source file
source	undo what spotless did
tags	run ctags, (using the -t flag is suggested)
rdist	distribute sources to other hosts
file.c	check out the named file from revision control

In addition, command-line defines can be given to define either Makefile values (such as ''CFLAGS'') or values in the program (such as ''DEBUG'').

---

11. Flag names may vary.

21. 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 each project program administration group.

- 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.
- If a project establishes its own archive libraries, it should plan on supplying a lint library file [2] to the system administrators. The lint library file allows *lint* to check for compatible use of library functions.
- What kind of revision control needs to be used?

22. Conclusion

A set of standards has been presented for C programming style. Among the most important points are:

- The proper use of white space and comments so that the structure of the program is evident from the layout of the code. The use of simple expressions, statements, and functions so that they may be understood easily.
- To keep in mind that you or someone else will likely be asked to modify code or make it run on a different machine sometime in the future. Craft code so that it is portable to obscure machines. Localize optimizations since they are often confusing and may be ''pessimizations'' on other machines.
- Many style choices are arbitrary. Having a style that is consistent (particularly with group standards) is more important than following absolute style rules. Mixing styles is worse than using any single bad style.

As with any standard, it must be followed if it is to be useful. If you have trouble following any of these standards don't just ignore them. Talk with your local guru, or an experienced programmer at your institution.

References

[1] B.A. Tague, *C Language Portability*, Sept 22, 1977. This document issued by department 8234 contains three memos by R.C. Haight, A.L. Glasser, and T.L. Lyon dealing with style and portability.

[2] S.C. Johnson, *Lint, a C Program Checker*, USENIX UNIX† Supplementary Documents, November 1986.

[3] R.W. Mitze, *The 3B/PDP-11 Swabbing Problem*, Memorandum for File, 1273-770907.01MF, September 14, 1977.

[4] R.A. Elliott and D.C. Pfeffer, *3B Processor Common Diagnostic Standards- Version 1*, Memorandum for File, 5514-780330.01MF, March 30, 1978.

[5] R.W. Mitze, *An Overview of C Compilation of UNIX User Processes on the 3B*, Memorandum for File, 5521-780329.02MF, March 29, 1978.

[6] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice Hall 1978, Second Ed. 1988, ISBN 0-13-110362-8.

[7] S.I. Feldman, *Make — A Program for Maintaining Computer Programs*, USENIX UNIX Supplementary Documents, November 1986.

[8] Ian Darwin and Geoff Collyer, *Can't Happen or /* NOTREACHED */ or Real Programs Dump Core*, USENIX Association Winter Conference, Dallas 1985 Proceedings.

[9] Brian W. Kernighan and P. J. Plauger *The Elements of Programming Style*. McGraw-Hill, 1974, Second Ed. 1978, ISBN 0-07-034-207-5.

[10] J. E. Lapin *Portable C and UNIX System Programming*, Prentice Hall 1987, ISBN 0-13-686494-5.

[11] Ian F. Darwin, *Checking C Programs with lint*, O'Reilly & Associates, 1989. ISBN 0-937175-30-7.

[12] Andrew R. Koenig, *C Traps and Pitfalls*, Addison-Wesley, 1989. ISBN 0-201-17928-8.

†UNIX is a Trademark of Bell Laboratories.

The Ten Commandments for C Programmers

Henry Spencer

1 Thou shalt run *lint* frequently and study its pronouncements with care, for verily its perception and judgement oft exceed thine.

2 Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end.

3 Thou shalt cast all function arguments to the expected type if they are not of that type already, even when thou art convinced that this is unnecessary, lest they take cruel vengeance upon thee when thou least expect it.

4 If thy header files fail to declare the return types of thy library functions, thou shalt declare them thyself with the most meticulous care, lest grievous harm befall thy program.

5 Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest ‘‘foo’’ someone someday shall type ‘‘supercalifragilisticexpialidocious’’.

6 If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest ‘‘it cannot happen to me’’, the gods shall surely punish thee for thy arrogance.

7 Thou shalt study thy libraries and strive not to re-invent them without cause, that thy code may be short and readable and thy days pleasant and productive.

8 Thou shalt make thy program’s purpose and structure clear to thy fellow man by using the One True Brace Style, even if thou likest it not, for thy creativity is better used in solving problems than in creating beautiful new impediments to understanding.

9 Thy external identifiers shall be unique in the first six characters, though this harsh discipline be irksome and the years of its necessity stretch before thee seemingly without end, lest thou tear thy hair out and go mad on that fateful day when thou desirest to make thy program run on an old system.

10 Thou shalt foreswear, renounce, and abjure the vile heresy which claimeth that ‘‘All the world’s a VAX’’, and have no commerce with the benighted heathens who cling to this barbarous belief, that the days of thy program may be long even though the days of thy current machine be short.