Recommended Coding Standards at Emertxe

Introduction :

The intension of having a coding guidelines is to have:

Code quality - Having the code in a uniform pattern so that it is easily readable by every one working in a same organization.

Productivity - Your code will be used, read, extended, interfaced-to, modified, documented, and ported by many other people, so it is important that every one use a consistent coding style. Good style should encourage consistent layout, improve portability, and reduce errors.

Ultimately, the goal of these standards is to increase portability, reduce maintenance, and above all improve clarity.

``*To be clear is professional; not to be clear is unprofessional.*" -- Sir Ernest Gowers.

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 80 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.

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.

- Any project should have a main file along with its header file named as "main.*" where * is the file extension

eg:

main.c

main.cc,

main.h, etc.,.

- The associated file should be named as based on its application with appropriate extensions

eg: If you are using i2c protocol then it should be named as

i2c.c

i2c.h, etc.,.

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.

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 judgment is called for here. If defining large numbers of essentially-independent utility functions, consider alphabetical order.

eg:

/*

* Author : Xyz

* Organization : Emertxe Information Technologies (P) Ltd.

* Date : 01-12-2006

* Usage : This is a example to show how to maintain a program file

*/

```
#include <stdio.h>
#include "main.h"

static int xyz;

static void increment()
{
xyz++;
}

void main(void)
{
increment();

return 0;
}
```

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. Header files should be self contained in header inclusions.

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.

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.

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.

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

Note that grep '^.\e*' will catch all block comments in the file. 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 */
}
```

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.

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 #defined 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 next 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 .

```
enum bt
{
e_ketch=1,
e_yawl,
e_sloop,
e_sqrig,
e_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 }, };
```

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 sSplodge_tT
{
    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

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 . Do not default to int; if the function does not return a value then it should be given return type *void* . 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, one space over. The function return type, name (and the formal parameter list) should be on a same line, one space after return type. 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.

If the function uses any external variables (or functions), they should come from some header file 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 *splint* will complain about it when given the --h option.

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);}
```

-- An example of code without whitespaces (decide how it is).

Use vertical and horizontal whitespace generously. Indentation and spacing should reflect the block structure of the code; e.g., there should be at least 1 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 (emertxe->next == NULL && totalcount < needed && needed <= MAX_ALLOT

&& server_active(current_input))

{ ...
```

Might be better as

```
if (emertxe->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.

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    skyblue()                                    /* true or false */
{
    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);
}
```

Simple Statements

There should be only one statement per line.

```
case                                                                BAS:
    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 e_false,
e_true
} 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.

As faor as possible avoid *goto*. *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. Function calls should always have a declaration before it, either explicitly or implicitly through its definition

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 the following standard.

```
control
{
    statement;
```

```
    statement;
}
```

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 splint-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 used and as the last one and does not require with 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.

Try to have an else always with an if

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

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.

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, on either side. 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.

Naming Conventions

- Names with leading and trailing underscores are reserved for system purposes and should not be used for any user-created names.
- *#define* constants should be in all CAPS.
- *enum* constants are *e_name* where *e* is for enum.
- 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 *emertxe* and *Emertxe*. Similarly, avoid *emertxebar* and *Emertxe_bar*. The potential for confusion is considerable. Use Hungarian notation for all typedefs, and classic C notation for all the remaining identifiers
- 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'.

The priority of using variables should be in the following order: local, static local, static global, global – local being the first. Globals, if used as a final resort, may alternativelyshould be grouped in a global structure. variable starting with g_.typedeffed names 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.

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. 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 floatdouble instead of 540 with an implicit floatdouble 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 as part of the standard I/O library's header file *stdio.h*. 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.

Macros

Complex expressions can be used as macro parameters, and operator-precedence problems can arise unless all occurrences of parameters have parentheses around them.

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 rewritten for thatsafely as: (Review this b4 accepting)

```
#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 splint
    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.

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 #ifdef'fed section may be scanned (processed) by the compiler, even if the #ifdef is false. Thus, even if the #ifdef'fed 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 forREALLOC 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*.

Debugging

"*C Code. C code run. Run, code, run... PLEASE!!!"* -- Barbara Tongue

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

```
enum
{
e_state_err,
e_state_start,
e_state_normal,
```

```
e_state_end
} State;

enum

{

e_val_new=1,

e_val_normal,

e_val_dying,

e_val_dead

} Value;
```

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 or use standard error functions like *perror* 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 – may be assert on such errors.

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.

Portability

"*C combines the power of assembler with the portability of assembler.*" *-- Anonymous, alluding to Bill Thacker.*

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. 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 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);Check the validity
- Note that the *size* of an object does not guarantee the *precision* of that object.
- 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.Never compare pointers with 0, rather use NULL.
- 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.

- ((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.
- Watch out for signed characters' default qualifier: signed vs unsigned – it is compiler dependent. Use explicitly, if needed.
- 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 #ifdef'fed or operations should be performed by #ifdef'fed 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, use the typedef "sized" types. Large programs should have a available in 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.
- Data *alignment* is also important.
- 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. Do not return structures rather return Sstructure 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 = emertxe(getchar(), getchar());

char emertxe(char c1, char c2, char 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 (see emertxetnote 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. 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.
- 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_t;
  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. Check the validity

- 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 emertxe(string) (printf("string = %s", (string)))
  ...
  emertxe(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. Check the validity
- Use *splint* 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 (*-Wallw* option to *gcc*). Treat all warnings as errors (-Werror), so that they are not ignored.
- 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, to make the desired result, explicit.
- 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.
- 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.

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. 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

```
#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.

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 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.

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 (indent). 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.) .
- 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.

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 *splint* complaints? A tolerance level needs to be established in concert with the *splint* 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 *splint* to check for compatible use of library functions.
- What kind of revision control needs to be used?