C Coding Guidelines

Revision 2

7 January 1993

John Reekie
School of Electrical Engineering
University of Technology, Sydney
PO Box 123
Broadway NSW 2007
AUSTRALIA

email: johnr@ee.uts.edu.au

Contents

Introduction

This document is a set of coding guidelines for the C programming language. The purpose of guidelines such as these is to improve:

- Programmer productivity, by releasing the programmer from the burden of making trivial but time-consuming style decisions.
- Code quality, by encouraging programming habits that avoid potential pitfalls.
- Maintainability, by encouraging an easy-to-read, consistent code format.

As an individual, guidelines provide you with a set of coding style and program structuring rules selected from any number of alternatives. At an individual level, coding style is often a matter of personal taste, so as you become more proficient in C, you can choose to apply different guidelines to your code.

At the level of an organisation, guidelines provide a consistent coding style for all members of the organisation. In this situation, your code will most likely be used, read, extended, interfaced-to, modified, documented, and ported by many other people, so it is important that a consistent style be enforced across the organisation. See (Strakker 92) for a discussion on the need for guidelines and standards.

This document is one of two companion documents, written mainly for use in an educational environment. This one deals only with coding style. The guidelines are presented as *recommendations*, not as a "standard," although they can be adopted as a standard if appropriate.

The other document, *Notes on C Programming*, requires a degree of familiarity with C to be useful, and deals with more complex issues such as program structure, assertions, debugging, and portability.

Note that this document does not attempt to teach C; nor does it deal with program design or documentation, except as manifested in the C source code.

Which C?

The only C sanctioned by this document is ANSI C, with a couple of minor exceptions. ANSI C compilers are now quite widely available, and standardising on ANSI C ensures a good level of portability.

These guidelines allow or require some deviations from the ANSI standard:

1. The ANSI standard only guarantees six non-case-sensitive significant characters in external identifiers, to ensure backwards compatibility with older linkers. Requiring all names to be unique in their first six characters would destroy program readability, so we ignore this restriction: these guidelines allow external identifiers can be up to 31 characters, and are case-sensitive.
2. Function prototypes must be used. This is a safety issue. While ANSI C adds function prototypes to C, it does not require that prototypes be used. Again, this is for compatibility reasons. Because omitting prototypes can even stop code from working properly, set the option on your compiler that causes the compiler to issue an error or warning message if a function without a prototype is called or defined.
3. Old-style function prototypes and definitions may not be used.

As much as possible, do not use any other non-ANSI extensions. Of course, many programs require platform-specific functionality, particularly with regard to user interfaces and interfacing to external devices—see *Notes on C Programming* for recommendations.

The “Golden Rules”

Scattered through this document are boxes highlighting the most important “rules.” The following four rules summarise the whole document—if you understand them, the rest of the document is mere detail.

The golden rule
Keep it simple.

Write simple, easy code. Put one statement per line. Don’t use a complex language feature when a simple one will do. Don’t use a complex algorithm when a simple one will do.

The other golden rule
Don’t be clever.

Do not try and take advantage of C’s “powerful” features unless you really know what you are doing. Attempts to do so often end in frustrated failure.

The clear coding rule
Write your code for people, not the computer.

Make your code readable to other people. You also will be able to read it better, and be far less likely to make errors.

The even-more-important clear coding rule
Be explicit

Write code so that its meaning is immediately clear. In other words, make it obvious. This rule is the one that answers to remarks such as “But everyone knows that ...,” naming one of C’s unique features. In actual fact, not everyone knows it. C programs are not always read or maintained by expert, practicing, C programmers, so don’t expect them to understand neat tricks you have perhaps only just learned yourself!

And finally, don’t forget that this document is a set of guidelines, not hard-and-fast rules—what really counts is that your code is correct, and easy to write, understand, and maintain.

Acknowledgments

Much of the material in this guide is the collected knowledge of many other programmers. The *References* sections lists a number of books and articles that contain more detailed or specialised guidelines.

I’m always looking for suggestions that will improve this document—or the *Notes on C Programming*, for that matter. If you have any, please send them by electronic mail to John Reekie at johnr@ee.uts.edu.au.

This document has been improved greatly by comments and suggestions from Chris Drane, Ruben Gonzalez, David Holmes, Rick Jelliffe, Craig Scott, and Peter Yardley.

Indentation and layout

Good layout is an important factor in code readability, and can also help avoid bugs. This section gives examples of the recommended style, with some advice on usage. There are two key points:

1. Indent each nested piece of code by one tab stop. Set tabs at FOUR (4) (*four*) spaces. If your editor doesn't use tabs, indent by FOUR (4) (*four*) spaces.
2. Opening and closing braces (with one exception) must be on a line by themselves.

An alternative style is described later.

while

```
while ( item != NULL )
{
    item->referenced = TRUE;
    item = item->link;
}
```

do-while

```
do
{
    read_buffer( input, buffer );
    process_buffer( buffer );
} while ( data_available( input ) );
```

do-while loops are used much less frequently than *while*-loops. Remember that a *do-while* loop will always be executed at least once, whereas a *while*-loop may not be executed at all.

for

```
for ( radix = MIN_RADIX; radix < ADDRESS_BITS; radix++ )
{
    pool = heap->pools + radix;

    pool->free_list     = NULL;
    pool->block_size    = (ulong)1 << radix;
    pool->size_radix    = radix;
}
```

Usage of *for*-loops

The syntax of C's *for*-loops is unusual, since it allows the *for* statement to contain completely arbitrary statements. Use *for*-loops only for loops that iterate a fixed number of times, based on the value of a counter. The code above is a typical example.

If the loop is *not* one that has a readily identifiable counter, then use a *while*-loop.

Multiple statements

It is possible to put multiple statements in each section of a *for* statement:

```
for ( i = 0, x = 0; i < limit; i++, x += buffer[i] );
```

Do not do this. Use only a single loop variable (counter), and use each section of the *for* statement to i) initialise the counter; ii) test for loop termination; and iii) update the counter. Any other operations should go before or inside the loop. Here is the above loop written in the recommended way:

```
x = 0;
for ( i = 0; i < limit; i++ )
    x += buffer[i];
```

Note how the counter *i* ranges from zero to *one less than the number of times through the loop.*

if-else

A single statement following an *if* requires no braces. Always put the statement on a new line:

```
if ( size < MAXIMUM_SIZE )
    size++;
```

Enclose each block of statements after an *if* or an *else* with braces:

```
if ( x == y )
{
    printf( "a equals b\n" );
    expand_block( x );
}
else if ( x > y )
{                       ← Note braces for consistency
    resize_block( x, y );
}
else
    ;   /* nothing */
```

Braces can be used around single statements for consistency. They also ensure that a bug is not inadvertently introduced at some later time by adding a new statement to a block. Note here the empty *else* clause—this helps someone reading the program later to realise that there are times when none of the branches will be taken. (This is similar to always putting a *default* case in a *switch* statement.)

The dangling *else* problem

There is a notorious bug caused by careless layout. The code on the left below does not behave as it looks:

<table>
<tr><th>Bad</th><th>Good</th></tr>
<tr><td><pre>
if (x < y)
    if (z != 0) x = z;
else
    x = y;
</pre></td><td><pre>
if (x < y)
{
    if (z != 0)
        x = z;
}
else
{
    x = y;
}
</pre></td></tr>
</table>

In fact, the code on the left actually executes as though it was this:

```
if ( x < y )
{
    if ( z != 0 )
        x = z;
    else
        x = y;
}
```

That is because an *else* associates with the closest unmatched *if* within the same pair of braces. The code should have been written as shown on the right above.

The nested *if* rule
Enclose all nested if*'s (except perhaps the innermost) in braces.*

switch

```
switch ( ch )
{
    case '\n':  /* Fall through */
    case '\r':
        token = NEWLINE;
        break;

    case '\t':
        token = TAB;
        break;

    default:
        token = ch;
        break;
}
```

Defaults

The *default* case is not enforced by C. Even if it does nothing, include it anyway, so that someone reading the program immediately sees that there are cases that are not processed:

```
    default:
        ;        /* otherwise, do nothing */
}
```

Often, not matching any of the cases indicates an error condition, so you can use the *default* case to signal an error.

Fall-throughs

The C *switch* construct allows a case to “fall through” to the next case: in the above example, the *'\n'* case falls through to the *'\r'* case. Fall-throughs must always be commented.

Use fall-throughs only for cases that share exactly the same code—that is, when there are no statements before the fall-through. Here is a bad example:

```
    ...
    case '\n':
        process_extra( '\n' );
                    /* fall through */
    case '\r':
        token = NEWLINE;
        break;
    ...
```

The problem with this approach is that a *switch* statement using fall-throughs can rapidly become as difficult to understand and modify as unrestricted use of *goto*'s. Although it requires repetition of some code, the above code should instead be written like this:

```
    ...
    case '\n':
        process_extra( '\n' );
        token = NEWLINE;
        break;

    case '\r':
        token = NEWLINE;
        break;
    ...
```

Conditionals

If the conditional statement spans more than one line, indent the second and third lines:

```
current = buffer->isready
          ? buf_read( buffer )
          : 0.0;
```

Null statements

A null statement is a statement that does nothing. Always put a null statement on a line by itself, together with a "/* nothing */" comment: Null statements are used mainly as empty bodies for *for*- and *while*-loops:

```
while (*p++ != '\n')          /* scan to end of line */
    ;   /* nothing */
```

Null statements may be used in *if-else* sequences to explicitly indicate a case for which nothing happens.

```
if ( empty( buffer ) )
    transfer_buffer( buffer );
else if ( isfull( buffer ) )
    ;       /* nothing */
else
    restart_buffer( buffer );
```

Never, ever, put a semi-colon immediately after an *if*, *for*, or *while* statement. At first glance, the following code simply looks like the second line has not been indented properly:

```
while (*p++ != '\n');
*p = '\0';
```

The null statement rule
Put every null statement on a line by itself, and comment *it.*

Line breaks

Break statements into multiple lines where necessary, indenting the second and following lines. Break just before an operator (so the operator goes on the next line), and line the operator up with the start of the expression on the first line. Following are some examples.

- Use braces to clearly separate a multi-line conditional test from following code:

```
if ( driver->status != Active
     && max_size( driver->idle_ports ) < MAX_READY
     && wgetc( front_window ) != \n )
{
    re_init( driver );
}
```

- Break arithmetic expressions at operators with weaker precedence:

```
x = sqrt( x ) * sin( 4.563 * y )
    + sqrt( 0.5 * v / ( 4.5 / w ) ) * 3.2;
```

Whitespace

There is only one thing to remember about whitespace: it's free, so use plenty of it. Typically, binary operators (+, –, etc) should be surrounded by spaces. Put spaces on the inside of parentheses. Insert lots of blank lines.

The whitespace rule
Whitespace is free—use it!

Comments

Good commenting is essential. Comments need to say enough that someone reading the program can understand it, but must not obscure the code.

The comment rule
Comment, but don't over-comment.

There are two styles of comment: block comments, and in-line comments.

Block comments

Block comments are on separate lines from code. They are indented to the same level as the code they are in (or the left edge if they are outside any function definition). There is *only one* format for block comments:

```
/*
 * This is a block comment. The first slash (/) must be
 * at the same level of indenting as the surrounding code
 * (or at the start of the line for comments not inside
 * any code).
 *
 * Leave a blank line between paragraphs in a block comment.
 */

/*
 * Incidentally, block comments can be as short as one line.
 */
```

Use block comments anywhere an explanation longer than a few words is needed:

```
    /*
     * Process the next sample from the buffer. If the
     * end of the buffer is reached, reset the index to
     * zero.
     */
    process( buffer[index++] );
    if ( index == buffer_size )
        index = buffer_start;
```

Do not use box-style comments. They clutter your code and are difficult to edit:

```
    /************************************************
     *  YUK YUK YUK YUK YUK YUK YUK YUK YUK YUK YUK *
     *  YUK YUK YUK YUK YUK YUK YUK YUK YUK YUK YUK *
     ************************************************/
```

Some alternative block comment styles are described later.

In-line comments

In-line comments go at the end of a line of code (and occasionally inside the code). In-line comments must be short and sharp:

```
    if ( x > limit )
        adjust_threshold( x );  /* deal with overflow */
    else
        process_normally( x );  /* usual case */
```

In-line comments must add explanation, not just duplicate the code. Comments like the following are useless:

```
    x += 1;          /* add one to x */
```

Don't put a comment at the end of every line of code. If you do, you are probably not explaining enough in block comments. Don't bother too much about lining comments up, since they'll just move again when the code is modified.

Names

Choosing names

The names of functions, variables, types, structure fields, constants, and macros must:

1. Reflect their purpose.
2. Be as concise as possible.

This document recommends the traditional C naming convention for most names: use only lower-case characters, with whole words separated by underscores ("_").

Table 1 lists different types of name and gives some examples. In general, names tend to be shorter the more frequent and localised their use. For more information on particular cases, see *Typedefs* and *Macros and constants*.

Table 1. Types of function and variable names

Type	Examples	Usage
Very short	i, j, k, p, q, ch	Local variables: *i*, *j*, and *k* for counters; *p* and *q* for pointers; *ch* for characters. These names must be used only in short functions.
Short	mkint, remove, isempty, size, maxlength	Commonly-used functions, such as those in widely-used libraries; local functions; local variables; structure fields
Long	allocate_memory, lookup_max, free_queue	Less commonly-used functions; functions in re-usable libraries; global variables
Capitalised	Queue, PortStatus, SymbolTable, BufferLength, DefaultHeap	Abstract data type names; enumerated constant type names and constant values; constants; variables private to a module
Upper-case	HASH_NAME, ALLOC_PTR	Macros

Function names

Here are a couple of tips for choosing function names:

- Start functions that perform an action with a verb: *empty_buffer*, *read_input*, *move_mountains*.
- Use words such as *is* and *has* in functions that return a boolean: *buffer_isempty*, *has_permission*, *within_range*.

When designing re-usable modules, adopt conventions about the meanings of certain words. For example, always use the word *empty* as a test, and the word *clear* as the equivalent verb—the difference between *buffer_empty* and *clear_buffer* is thus clear.

Functions in libraries of modules should conform to naming conventions that clearly identify the module or class of function to which each function belongs. The *Notes on C Programming*, for example, recommends that functions belonging to ADTs be prefixed with a three- or four-letter identifier code. This can make names hard to pronounce, but is essential for minimising name clashes.

Underscores

Many compilers use names beginning or ending with underscores for their own internal use. Do not use names that begin or end with an underscore.

Do not use double underscores: they look much the same as single underscores when printed.

Over-long names

Some programming texts take the advice to give functions and variables meaningful names a little too far: a name such as *maximum_or_minimum_divisor* is too long. Although it seems meaningful, names like this quickly become tedious to write and read.

Macros and constants

Macros are a textual substitution tool. The C pre-processor expands macros before the code is compiled.

Macro names

As a general rule, make macro names CAPITALISED, and use underscores between words in macro names. This ensures that macros are readily identifiable as such.

The C standard library contains some macros with lower-case names, so that they look like functions. In general, this should be avoided. If you do have macros with non-capitalised names, make sure that the macro uses each argument *only once* (see *Side-effects*).

Constants

Using *const*

In ANSI C, constants can be declared with the *const* keyword. For example:

```
const float pi = 3.1415926535;
const int BufferLength = 512;
```

Constants eliminate “magic numbers”—numbers that appear in code without any apparent reason for being some particular value. Magic numbers make it very difficult to modify code, because it is difficult or impossible to tell which occurrences of a particular number need to modified:

```
while ( i < BufferLength )       /* GOOD */
    ...
while ( i < 512 )                /* why 512? */
    ...
```

Use enumerated constants for series of “flag” type constants (see *Typedefs*).

Using macros

Macros were traditionally used to define constants:

```
#define BUFFER_LENGTH 512
```

The newer *const* keyword is preferred for constants that are valid C types, such as numbers. However, macros are still useful for textual substitution. They also have the advantage that they can be defined when the compiler is invoked. For constants that control program configuration, the following sequence is useful:

```
#ifndef BUFFER_LENGTH
#   define BUFFER_LENGTH 512
#endif
```

If the program is compiled with *BUFFER_LENGTH* specified to the compiler (for example, by *cc -dBUFFER_LENGTH T =256 xxx.c*), then the program will be compiled with the specified value, otherwise it will be compiled with the default value of 512.

Constant pointers

The syntax for declaring constant pointers is slightly confusing. The following is a declaration of a pointer to a hardware peripheral. It is declared as a constant to ensure that the pointer is never changed.

```
port_t const *Port0 = (port_t const *) 0x808040;
```

Note the use of a typecast to convert the hardware address into a C pointer. Be careful not to confuse your declaration with that for declaring a pointer to a constant (rather than a pointer which *is* constant, but points to a non-constant value):

```
const port_t *ThisIsSilly;
```

Macros as expressions

Macros can be used to define expressions—that is, they return a value. For example, this macro converts degrees to radians:

```
#define DEG_TO_RAD(x) ((x) * 3.1415926535 / 180)
```

Always enclose the macro result in parentheses. Without the enclosing parentheses, this:

```
    x = y / DEG_TO_RAD( phase );
```

would expand to the (obviously incorrect) code:

```
    x = y / phase * 3.1415926535 / 180;
```

Macros as statements

If a macro is intended to be used as a statement—that is, it *does* something rather than return a result—then enclose the statements in the macro in a *do-while* loop with a *FALSE* condition:

```
#define MARK_STACK() \
    do \
        { StackMarker = 1; StackMarker += ObjectSize } \
    while( FALSE )
```

This use of a *do-while* loop to enclose the statements in the macro is especially important for macros that contain *if-else* statements or *while*-loops.

Note that there is no terminating semi-colon in the macro definition: this is inserted by the caller. *Never* put a terminating semi-colon in any macro definition.

Parenthesising arguments

In macro definitions, enclose every use of a macro argument in parentheses. This avoids problems with operator precedence when the macro is expanded. For example, the following code:

```
    x = DEG_TO_RAD(phase + delta);
```

expands to

```
    x = ((phase + delta) * 3.1415926535 / 180); /* OK! */
```

Without the parentheses around *x* in the definition of *DEG_TO_RAD*, the macro would expand to this (obviously incorrect) code;

```
    x = (phase + delta * 3.1415926535 / 180);    /* Broken! */
```

The macro parenthesis rule
Parenthesise everything!

Commenting

Comment each macro as well as you would comment a function. In particular, indicate whether the macro is intended to return a value (as in *DEG_TO_RAD*), or to operate like a procedure (as in *MARK_STACK*).

Other cautions

Empty argument lists

If a macro takes no arguments, give it an empty argument list. This forces the caller to insert parentheses, and makes it clear to the reader that the macro is not just a constant. For example, the MARK_STACK macro above is called like this:

```
..
MARK_STACK();
..
```

Space after macro name

Do not insert a space between the macro name and the first parenthesis. this:

```
#define BOO (x) (x+2)
```

is actually a constant *BOO* defined to expand to *(x) (x+2)*!

Side-effects

When calling macros, do not provide arguments that can generate side-effects, in case the macro uses the argument twice:

```
#define DOUBLE(x) ((x)+(x))
```

The following code:

```
x = DOUBLE(*p++);
```

← *This will increment p TWICE!*

expands to

```
x = ((*p++)+(*p++))
```

which is obviously not the intended effect (see *Side-effects*).

Misuse and abuse

Macros are a powerful feature—but don't over-use them. Avoid writing excessively complex macros—they can be unreliable, and can make your code hard to read. They can also make it difficult to use a debugger—use a function call instead.

Also, don't use macros to define new syntax. One example I have seen attempts to make C look like Pascal. This is highly error-prone, and can make your code quite unreadable.

Comparison tests

The NULL pointer

When comparing a pointer with NULL, write it that way. This makes code clearer, and helps you avoid logical errors (like forgetting to insert the '!' operator).

```
if ( !port->buffer )                        /* BAD */
    ...

if ( port->buffer == NULL )                 /* GOOD */
    ...
```

Boolean tests

C does not have a boolean variable type. Use the *bool* type defined in *stdhdr.h* (see Appendix) to make your programs easier to read. (Note that it does not add true boolean variables to C). Avoid unnecessarily using other pairs of names with values zero and one (such as YES/NO, ON/OFF, SET/RESET, and so on).

Declare and initialise a boolean variable like this:

```
    bool interrupted = FALSE;
```

Use either of the following two methods to set a boolean variable. The second is more "logical," but the first may be easier to read.

```
    if ( x > y )
        interrupted = TRUE;

    interrupted = (x > y);
```

Test boolean variables without using TRUE and FALSE.

```
    if ( interrupted == TRUE )              /* BAD */
        ...
    if ( interrupted )                      /* GOOD */
        ...
```

Do not test non-boolean variables as though they were:

```
    int x;
    if ( x )                                /* BAD */
        ...
    if ( x != 0 )                           /* GOOD */
        ...
```

The null character

In C, character strings are terminated with a character with the value zero. This character can be written *'\0'*. If testing for a null character, explicitly test against this character.

The following example illustrates a common idiom that can hardly be labelled bad style—nonetheless, you should avoid using code like this until you are proficient at C:

```
    while ( *p++ )                  /* AVOID */
        ; /* nothing */

    while ( *p++ != '\0' )          /* PREFERRED */
        ;   /* nothing */
```

Side-effects

Side-effects are the general phenomenon in which an operation that returns a value also has some other effect. C constructs that have side-effects include assignment, the increment and decrement operators (++ and --), and update operations (x+=y). Side-effects are a notorious and copious source of bugs, and so deserve special attention.

In this section, the word “assignment” refers to operators like “+=” and “/=” as well as “=”.

Assignment in conditional tests

Do not use assignment in the test clause in *if-then-else*, *switch*, conditional (*?:*), *while*, or *do-while* statements. Of itself, assignment in this position is not a bug—however, it will cause one if parentheses are inadvertently omitted.

```
if ( (block = malloc( BLOCK_SIZE )) != NULL ) /* AVOID */
    ...

block = malloc( BLOCK_SIZE );                 /* GOOD */
if ( block != NULL )
    ...
```

Some well-known idioms do make coding rather easier: use them only if you are an experienced programmer, otherwise avoid them. In any case, the preferred example is probably better from a long-term maintenance view-point.

```
while ( (ch = getchar()) != '\n' )          /* AVOID */
{
    ...
}

ch = getchar();                             /* PREFERRED */
while ( ch != '\n' )
{
    ...
    ch = getchar();
}
```

Other assignment traps

A good general rule is not to mix assignment with comparison tests, and to have only one assignment in a statement. Here are some potentially troublesome examples.

```
x += (x == y);            /* BAD */

if (x == y)               /* GOOD */
    x++;
```

The following bad example does not in fact do the same thing as the good version:

```
if (x=y < z)              /* BAD */
    ...

x = y;                    /* GOOD */
if (x < z)
    ...
```

The following bad example is inviting the conditional to be “fixed” to *if (x == y)*:

```
if ( x = y )
    ...

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

Many of the above rules are summarised by this rule:

The assignment rule
Do not use assignment as an expression.

In other words, ONLY use assignment as a statement, on a line by itself.

Increment operators

The C increment and decrement operators ++ and -- are very powerful. But they are not as simple as they look. If a variable is used more than once in the same expression or statement, and is incremented or decremented (or assigned to) one or more times, then the result is *UNDEFINED*.

For example:

```
*p++ = *p - x;       /* BAD -- Won't necessarily work! */

*p = *p - x;             /* GOOD */
p++;

*p++ = *q++;             /* This is fine */
```

Here is a similar problem:

```
xs[i++] = i;
```

Sometimes, this will assign the value of *i* to the *i*th element of the array *xs*; sometimes it won't.

The side-effect rule
Do not side-effect any variable used more than once in an expression.

It is a good general rule to use side-effects only in expressions that contain only one variable.

Function calls

Order of evaluation of function arguments is *UNDEFINED*. Never use side-effects in arguments to functions:

```
rotate( *p++, *p++ );        /* BAD -- what will it do? */

rotate( *p, *(p+1) );        /* GOOD */
p += 2;

writeoff( x *= 2, sqrt(x) );    /* BAD */

x *= 2;                         /* GOOD */
writeoff( x, sqrt(x) );
```

Be careful!

Just because code such as

```
*p++ = *p - x;
```

works as you expect when you try it, that does *not* mean that it is “OK.” Any code the effect of which is undefined will “work” on some computers and compilers, and not on others. Ignoring these warnings because the code “works” is asking for trouble later!

Exits and goto's

Early exits

It is sometimes claimed that functions should have only a single exit point, at the end of the function. However, this has drawbacks:

- It can lead to unnecessarily deep, and therefore difficult to follow, block nesting.
- It can make it difficult to figure out exactly what the return value is if an error occurs, and what code gets executed under different conditions.

On the left below is an example function written this way. With more error cases, the nesting gets much worse.

<table>
<tr><th>Not recommended</th><th>Recommended</th></tr>
<tr><td>

```
p = malloc( BUF_SIZE );
if ( p != NULL )
{
    q = malloc( BUF_SIZE );
    if ( q != NULL )
    {
      ../* process p and q */
    }
}
return p; /* NULL if error */
```

</td><td>

```
p = malloc( BUF_SIZE );
if ( p == NULL )
    return NULL; /* error */

q = malloc( BUF_SIZE );
if ( q == NULL )
    return NULL; /* error */

... /* process p and q */

return p;  /* valid value */
```

</td></tr>
</table>

These guidelines recommend against this style. Rather, if an error occurs, it should be handled immediately, and the function exited with an error code if appropriate. This is shown on the right above.

Goto

The *goto* statement has had a lot of bad press. Like global variables, *goto*'s should be used infrequently and with great care, but they do have their place.

The only use of *goto* sanctioned by these guidelines is to branch to error-handling code at the end of a function. This should be done only when the error-handling code is non-trivial, and inserting it where the error occurred would be clumsy and repetitive.

```
    p = malloc( BUF_SIZE );
    if ( p == NULL )
        goto abort;     /* error */

    if ( ... )
    {
        while ( ... )
        {
            if ( something_really_serious_happens() )
                goto abort;     /* error -- jump to clean-up */
        }
    }
    return p;                   /* normal return */

abort:                          /* error clean-up and return */
    if ( p != NULL )
        free( p );
    ...

    return NULL;                /* error return status */
}
```

Note the defensive *if (p != NULL)* test in the clean-up code.

Typedefs

Type definitions (typedefs) define new types, and help make a program more readable and safer. Typedefs on simple types (integers, etc) should generally be avoided.

Structures

Always define a new type when defining a structure. For structures that do not implement an abstract data type (see *Notes on C Programming)*, the structure typedef should look like this:

```
typedef struct complex_s
{
    double  real;
    double  imag;
} complex_t;
```

Note the use of the suffixes "_s" to mark the structure tag, and "_t" to mark the type tag. Although this naming convention may seem awkward at first, it makes it much easier to think up and remember type names.

Field names

Structure field names should be "short" names (see *Names*) if possible. As long as structure variables or pointers are well-named, short names do not lead to ambiguous code. For example,

```
    Window frontwin;
    ...
    frontwin->top = frontwin->bottom + frontwin->depth.limit;
```

Using long names for fields just obscures the meaning:

```
    frontwin->window_top = frontwin->window_bottom
                         + frontwin->window_depth.depth_limit;
```

Enumerated constants

Use enumerated constants to give names to a set of "flag" values:

```
#define PORT_IDLE   0               /* BAD */
#define PORT_READY  1
#define PORT_ACTIVE 2

typedef enum                        /* GOOD */
{
    PortIdle,
    PortReady,
    PortActive
} PortStatus;
```

Enumerated constants have the first letter of each word capitalised. Often, the second word of the type name will be something like "Type" or "Status" or "Mode." The first item of an enumerated type should indicate an error or inactive status.

Because C does not keep separate name-spaces for each enumerated constant type, all enumerated constants in a program must be unique. For this reason, prefix each constant with a word (in this case, *Port*) indicating the type the constant belongs to.

Avoid setting enums to particular values:

```
typedef enum
{
    Invalid,
    Integer,
    Float = 7,          /* rarely necessary */
    Complex
} DatumType;
```

Definitions, declarations, and prototypes

C has two types of file: header files, with names ending in “.h”, and source code files, with names ending in “.c”. They serve two very different purposes, which must not be confused:

- Source code files contain functions and variables.
- Header files contain declarations of functions and variables.

Header files must *never* define functions or variables. They are *included* by other header files and source code files to inform the compiler of the types of variables and functions defined in other files.

See *File structure* for a summary of header and source code file structure.

Function prototypes

A function prototype gives the name of a function, the types of its arguments, and its return type. Prototypes of globally-visible functions are put into header files; prototypes of local (static) functions are contained in the same file as the function. Here is an example of a prototype:

```
void
free_memory( pointer mem, size_t size );
```

Note that

i) The function name starts on a separate line from its return type. This makes it easier to find the function name.

ii) The names of the arguments are given, as well as their types. This allows you to refer to arguments by name in the comment that describes the function. It also saves some work: the prototype can be used as-is when writing the function implementation.

If you do not define a prototype for every external function, your code may not work as expected. Define a prototype for *every* function you write.

The prototype rule
Define a prototype for every function.

Function prototype comment

Precede every prototype by a block comment containing the function name and a description of its purpose, its arguments, and its return values. Use the following format:

```
/*
 *  q_push()
 *
 *  Put an item onto the front of the queue.
 *
 *  Returns:
 *      Nothing.
 *
 *  Asserts:
 *      'queue' is not NULL. 'item' is not NULL.
 */
void
q_push( Queue queue, pointer item );
```

The following fields must be included in every prototype comment:

Returns: Specify the return value of the function. If the function can return with an error, specify both the normal return value and the return value if an error occurs. Enter "Nothing" for *void* functions.

Asserts: List the assertions within the function. These assertions must match those in the function definition.

The following fields are optional:

Arguments: List each argument and its effect. This is required only if the purpose of each argument is not already obvious from the function description.

Errors: List program-detected errors. This field is required only for functions that take some action on an error condition—such as printing a message, aborting the program, or displaying an alert box.

Caveats: List additional conditions attached to use of the routine. This might include warnings about the effect of the routine or a warning that the routine can only be used in restricted circumstances.

Function definitions

A function definition is the implementation of a function. Function definitions can *only* be contained in source code files. Each function definition is preceded by a short block comment, which briefly describes its purpose. Often, the function definition comment need only be the first section of the header comment:

```
/*
 *  q_push()
 *
 *  Put an item onto the front of the queue.
 */
void
q_push( Queue queue, pointer item )
{
    ....;
}
```

In other cases, the implementation of the function may be quite complex—numerical algorithms, for example. Functions such as these need a longer comment, which succinctly describes the algorithm or mathematics involved.

Note that the function comment goes before the function definition, *not* between the function name and the function body, as in:

```
void
q_push( Queue queue, pointer item )
/*
 *  Put an item onto the front of the queue.
 */
{
    ....;
}
```

Variable declarations

Variables declared in header files must be prefixed with the keyword *extern*. This indicates to the compiler that this is a declaration of the variable, not the definition. A variable can be defined in only one source code file. For example, if the file *guru.c* defines the *char* pointer variable *cryptic_message*, then a header file, called say *guru.h*, will contain the declaration

```
    extern char *cryptic_message;
```

File structure

This section describes the recommended layout of files. See the Appendix for complete examples.

Header files (".h")

```
#ifndef _WindowManager_h_  —Prevent multiple inclusions—
#define _WindowManager_h_
/*
 *   WindowManager.h
 *
 *   The window manager module. This module ......
 *
 *   —Describe the purpose of the module and briefly how it works. Long—
 *   —detailed explanations go in the accompanying documentation.—
 *   .....
 *
 *   Changes:       —Change history, in reverse chronological order—
 *   27 Jul 92 -- Improved tiling routine -- John Reekie
 *   26 Jul 92 -- Created -- John Reekie
 */
#include "stdhdr.h"              —Always include this.—
#include <math.h>                     —Include other header files by this interface.—
#include "PaneManager.h"         —Standard headers first, then others.

—Macros used by module clients—

—Types and structures defined by this module—

—"Extern" declarations of global variables defined in this module—

—Prototypes for functions implemented by this module—

#endif /* WindowManager.h */
```

Source files (".c")

```
/*
 *   WindowManager.c
 *
 *   Implementation of the window manager module. See the file
 *   WindowManager.h and the WinLib documentation for more
 *   information.
 *
 *   —Note that this file header is quite short! There is no need to repeat—
 *   —information already in the header file.—
 */
#include "WindowManager.h"       —Include my own header.—

#include <string.h>         —Include other headers needed by this implementation.—
#include "SortedList.h"          —Standard headers first, then others.—

—Global variables, if any—

—Types needed only by the implementation—

—Define local (static) variables—

—Prototypes for local functions—

—Local and global definitions—

/* END */
```

Assertions

Introduction

An *assertion* specifies that a program satisfies certain conditions at particular points in its execution. In C, assertions are implemented with the standard *assert* macro. The argument to the *assert* macro must be true when the macro is executed, otherwise the program aborts and prints an error message. For example, the assertion

```
    assert( size <= LIMIT );
```

will abort the program and print an error message like this:

```
    Assertion violation: file tripe.c, line 34: size <= LIMIT
```

if *size* is greater than *LIMIT*.

Usage

Assertions are used to check the input conditions to a function. Two common uses are to assert that i) pointers are not *NULL*, and ii) indexes and size values are non-negative and less than a known limit. If an assertion is violated during program testing and debugging, there is a bug in the code that *called* the function containing the assertion. The bug must be found and fixed.

Here is an example of a function with assertions:

```
int
magic( int size, char *format )
{
    int maximum;

    assert( size <= LIMIT );
    assert( format != NULL );
    ...
```

Each assertion must be listed in the *Asserts* section of the function prototype comment. For example, the comment describing *magic* will include:

```
 *  Asserts:
 *      'size' is no greater then LIMIT.
 *      'format' is not NULL.
 */
```

If there are no assertions, write “Nothing”:

```
 *  Asserts:
 *      Nothing
 */
```

Assertions and error-checking

Assertions are a tool for finding program errors, not a run-time error-handling mechanism. An assertion violation caused by the user inadvertently entering a negative number when a positive number is expected is poor program design. Cases like this must be handled by appropriate error-checking and recovery code (such as requesting another input), *not* by assertions.

Turning assertions off

By default, ANSI C compilers generate code to check assertions at run-time. Assertion-checking can be turned off by defining the *NDEBUG* flag to your compiler:

```
    cc -dNDEBUG ...
```

This should be done only if you are confident that your program is operating correctly, and only if program run-time is a pressing concern.

Other traps

This section lists a number of other traps that are easy to fall into.

= and ==

One of the easiest mistakes to make in C is to use "=" (assignment) instead of "==" (equality). Because C treats assignment as an expression, code such as

```
if ( x = y )
```

is perfectly legal C (although not recommended by these guidelines!). This risk of this error can be reduced by placing constants on the left of the equality operator:

```
if ( NULL == p )
    ...
```

Inadvertently using an "=" here will cause a compile-time error. Other than that, the only solution is to double-check every conditional test. Following the practice of never using assignments in conditional tests, as recommended by these guidelines, will make it easier to recognise this error.

Operator precedence

C's operators do not always have the precedence you expect. Precedence errors can be particularly hard to find if you are not thoroughly familiar with all of C's operators, especially if the error is compounded by suggestive spacing. For example,

```
x = a + b<<1;
```

is really

```
x = (a + b) << 1;
```

Always check operator precedence if you are not *absolutely certain* of precedence. If in any doubt (and even if not, for the benefit of others who might read your code), use parentheses to make sure there is no mistake.

Allocating memory

Be careful of the argument to *malloc*. The following code is correct:

```
typedef struct something_s { ... } something_t;
...
something_t *p;
...
p = malloc( sizeof( *p ) );
```

The following is not—it allocates only enough space for the pointer to the structure:

```
p = malloc( sizeof( p ) );
```

To avoid this error, apply *sizeof* to the structure tag, not the pointer variable:

```
p = malloc( sizeof( struct something_s ) );
```

Implicit type conversion

C automatically converts between numeric types. For example, the result of a floating-point operation is converted to integer before assignment to *x* in the following:

```
long x;
float f;
...
x = f * pi;
```

To make it clear to other readers of your code that you *intend* that the floating-point value be converted to an integer, insert an explicit type-cast:

```
x = (long)(f * pi);
```

Alternative styles

Like most things in software engineering, there is no such thing as “correct” coding style. The style recommended in these guidelines is fairly widely-used, and recommended here because it is sparse and uncluttered. However, there are a number of other styles in common use.

This section presents some of these alternatives. Neither the one in the main body of this document or the alternative is inherently better then the other. Having said that:

- If you are writing code to be included in an existing library, or modifying existing code, *use the same style as the existing code.*
- If you are writing code for an organisation, use the style condoned or required by that organisation.

The Conformance Rule
Write your code the same as the code around it.

Other than that, coding style is largely a matter of personal taste. But whichever style you choose, *stick to it.*

An alternative indentation style

An alternative indentation style does not start a new line before the opening brace. For example:

```
if ( x == y ) {
    printf( "a equals b\n");
    expand_block( x );
}
else if ( x > y ) {
    resize_block( x, y );
}
else
    ;   /* nothing */
```

An alternative commenting style

An alternative block comment style is also widely used:

```
/*
**  This is a block comment. This style is sometimes preferred
**  for comments at the top level of code (such as comments
**  to functions and at the start of files) because the
**  double asterisk makes it stand out more.
*/
```

Many people prefer to separate whole functions more clearly, and so include a line of asterisks or dashes within the function definition and prototype comment, like so:

```
/*  -------------------------------------------------------
 *
 *  magic()
 *
 *  Blah blah blah blah. Presto magico ultimo jammo.
 */
```

Don’t use any more than this, since too many asterisks, dashes, and so on, just make it harder to find things, not easier.

Empty else and default clauses

The empty else clause, as in the above example, is usually omitted. Empty default clauses in switch statements are also often omitted. I have recommended they be inserted for a good reason: it is obvious to someone else reading the code that there is a case for which no code is executed. However, if you find this pedantic, you could omit these empty clauses:

```
if ( x == y ) {
    printf( "a equals b\n");
    expand_block( x );
}
else if ( x > y ) {
    resize_block( x, y );
}
```

An alternative naming style

This document recommends that function and variables names consisting of more than word be separated by underscores. An alternative style that seems to be gaining popularity does not use underscores, but capitalises the first letter of the second word onwards: *allocateMemory*, *lookupMax*, *freeQueue*, *largestFreeBlock*.

References

Peter Darnell and Philip Margolis, *C: A Software Engineering Approach*, Springer-Verlag, 1991.

A textbook containing “Alert Boxes” on common style and coding errors. Useful as a C text and reference as well as for coding guidelines.

Cannon, Elliot, et al, *Recommended C Style and Coding Standards*, 1990.

This is an updated version of the *Indian Hill C Style and Coding Standards*. Available by anonymous ftp from *cs.washington.edu* as *ftp/pub/cstyle.Z*.

Ian Darwin and Geoff Colyer, *Can’t Happen, or /* NOTREACHED */, or Real Programs Dump Core*.

Available by anonymous ftp from *cs.toronto.edu* in the *doc/programming* directory. UNIX-oriented.

Andrew Koenig, *C Traps and Pitfalls*, Addison-Wesley, 1989.

A short book containing advice on how to avoid most common errors made by C programmers.

Rob Pike, *Notes on Programming in C*.

Available by anonymous ftp from *cs.toronto.edu* in the *doc/programming* directory.

John Reekie, *Notes on C Programming*, 1993.

Notes on various aspects of C programming, such as program structure, assertions, debugging, and portability.

Henry Spencer, *How to Steal Code, or Inventing the Wheel Only Once*.

Available by anonymous ftp from *cs.toronto.edu* in the *doc/programming* directory.

David Strakker, *C Style: Standards and Guidelines*, Prentice-Hall, 1992.

An comprehensive collection of C coding guidelines. Includes a chapter on how to write and implement coding standards in an organisation.

Appendix

Standard header

The following header file can be included by every header file. Do not modify this file to contain program-specific typedefs and macros!

```
#ifndef _stdhdr_h_
#define _stdhdr_h_
/*
 *   stdhdr.h
 *
 *   Standard include file for ANSI compilers. Note that <stdio.h> is
 *   included only is USE_STDIO is defined.
 *
 *   Changes:
 *   23 dec 92 -- revised and simplified -- hjr
 *   5 june 92 -- first useful incarnation -- hjr
 */

#include <stdlib.h>        /* standard header files */
#include <stddef.h>
#include <assert.h>
#ifdef USE_STDIO
#   include <stdio.h>
#endif

/*
 *   "Boolean" typedef.
 */
typedef enum
{
    FALSE,
    TRUE
} bool;

/*
 * Aliases for double-barreled type names.
 */
typedef unsigned char      uchar;
typedef unsigned short          ushort;
typedef unsigned long      ulong;

/*
 * Typedef for the anonymous pointer type.
 */
typedef void *pointer;

/*
 *   debug(), ifdebug()
 *
 *   Macros for debugging. Note the double parentheses for debug:
 *
 *       debug( ("Received value %d from port %s", val, port->name) );
 */
#ifdef DEBUG
#   define debug(x)        printf x
#   define ifdebug(x)           x
#else
#   define debug(x)        /* nothing */
#   define ifdebug(x)           /* nothing */
#endif

#endif   /* _stdhdr_h_ */
```

Example header file

The following header file illustrates the layout of header files and function prototypes. Parts of the file have been deleted to save space.

```
#ifndef _Chain_h_
#define _Chain_h_
/*
 *   Chain.h
 *
 *   Low-level singly-linked list utilities.
 *
 *   A 'chain' is a linked list of items, in which the links are
 *   contained within the items themselves. Any given item can thus be
 *   contained in _only_ _one_ chain at any given time.
 *
 *   This module is intended for use in low-level system applications
in
 *   which elegance and protection are less important than efficiency.
For
 *   a more flexible, more general-purpose, and safer list module, use
 *   List.
 *
 *   In order to be generic, it is assumed that the very first element
of
 *   any item put into the chain is a pointer to the next item. Any
item
  * put onto the chain MUST have this field reserved for this
purpose.
 *   Chains can be sorted by the addresses of the items (reflecting
 *   the intended use in low-level system functions).
 *
 *   CHANGES
 *
 *   24 june 92 -- renamed Link to Chain, some tidying up -- john
reekie
 *   12 june 92 -- created -- John Reekie
 */
#include "stdhdr.h"


/*
 *   CLink
 *
 *   A link of the chain. This is a dummy structure used to make
 *   the code more readable -- the essential point is that the first
 *   word of any structure used as a chain item must be a pointer to
 *   the next item. A NULL pointer terminates the chain.
 */
typedef struct CLink_s
{
    struct CLink_s *next;
} *CLink;


/*
 *   chn_length()
 *
 *   Count the length of a chain.
 *
 *   Returns:
 *       The length of the chain. The length is zero if 'chain' is
NULL.
 *
 *   Asserts:
 *       Nothing.
 */
size_t
chn_length( pointer chain );
```

```
/*
 *   chn_sort()
 *
 *   Sort a chain, using the mergesort algorithm, combined with an
  *  insertion sort for short lists. Sorts in order of increasing
address.
  * Note that the return value _must_ be assigned to the chain
holder:
 *
 *       a_chain = chn_sort( chain );
 *
 *   Returns:
 *       The sorted chain. NULL if 'chain' is NULL.
 *
 *   Asserts:
 *       Nothing.
 */
pointer
chn_sort( pointer chain );


/*
 *   chn_cons(), CHN_CONS()
 *
 *   Put an item onto the front of the chain.
 *
 *   chn_cons() is a function. CHN_CONS() is a macro that expands into
 *   in-line code if NDEBUG is defined, otherwise into a call to
 *   chn_cons().
 *
 *   Returns:
 *       The new chain (actually the same as the new item). Note that
the
 *       return value _must_ be assigned to the pointer to the chain.
 *
 *   Asserts:
 *       chn_cons() only: 'item' is not NULL.
 */
pointer
chn_cons( pointer chain, pointer item );

#define CHN_CONS(c,i) INLINER( chn_cons(c,i), ( ((CLink)i)->next = c,
i ) )

#endif /* _Chain_h_ */
```

Example source file

The following source file illustrates the layout of source files and source code. Parts of the file have been deleted to save space.

```
/*
 *   Chain.c
 *
 *   Low-level singly-linked list utilities.
 */
#include "Chain.h"


/*
 *   NEXT
 *
  *  Macro to access the 'next' field of a chain. Just saves type
casting.
 */
#define NEXT(chn) ( ( (CLink) chn ) -> next )
```

```
/*
 *   local_merge_sort()
 *
 *   Sort a chain in order of increasing item address. This routine
 *   accepts the length of the chain as an extra argument.
 *
 *   Returns:
 *        The sorted chain.
 *
 *   Asserts:
 *        Nothing.
 */
static pointer
local_merge_sort( pointer chain, size_t length );


/*
 *   local_insertion_sort()
 *
 *   ....
 *   ....
 */
static pointer
local_insertion_sort( pointer chain, size_t length );


/*
 *   chn_length()
 *
 *   Count the length of a chain.
 */
size_t
chn_length( pointer chain )
{
     size_t   length;       /* the length of the chain */

     length = 0;
     while ( chain != NULL )
     {
          length++;
          chain = NEXT( chain );
     }
     return length;
}


/*
 *   chn_cons()
 *
 *   Put an item onto the front of the chain.
 */
pointer
chn_cons( pointer chain, pointer item )
{
     assert( chain != NULL );
     assert( item != NULL );

     NEXT(item) = chain;
     return item;
}
```

```
/*
 *   chn_sort()
 *
  *  Sort  a  chain  of  blocks  in  order  of  increasing  address.  For
efficiency,
 *   chn_sort() uses the auxiliary function local_merge_sort().
 *
 *   The merge sort is terminated when the chain length gets below
  * ARRAY_SORT_LIMIT,  and  an  insertion  sort  using  an  array  is
performed
 *   instead. With chains 20,000 elements long in essentially reverse
order
 *   (i.e. worst case), ARRAY_SORT_LIMIT equal to 16, on a 68030, the
  *  addition  of  the  insertion  sort  gave  about  50%  improvement  in
speed.
 */
#define ARRAY_SORT_LIMIT 16        /* 16 seems about optimal */

pointer
chn_sort( pointer chain )
{
    return local_merge_sort( chain, chn_length( chain ) );
}
```

```
/*
 *   local_merge_sort()
 *
 *   Sort a chain, but take its length as an argument.
 */
static pointer
local_merge_sort( pointer chain1, size_t length )
{
     CLink    item;         /* a temporary pointer to an item */
     pointer  chain2;       /* the second half of the chain */
     size_t   count;        /* a temporary counter */
     size_t   length2;      /* half the chain length (about) */


     /*
       *  Check  the  length:  if  less  than  ARRAY_SORT_LIMIT,  use  an
insertion
      *   sort.
      */
     if (length < ARRAY_SORT_LIMIT )
          return local_insertion_sort( chain1, length );


     /*
      *   Divide the chain into two (approximately) equal halves.
      */
     count = 1;
     length2 = length / 2; /* integer division? */

     assert( length2 >= 1 );

     item = chain1;
     while ( count < length2 )
     {
          item = item->next;
          count++;
     }
     chain2 = item->next;
     item->next = NULL;

     /*
      *   Recursively sort and then merge the two chains
      */
     chain1 = local_merge_sort( chain1, length2 );
     chain2 = local_merge_sort( chain2, length - length2 );

     return chn_merge( chain1, chain2 );
}
```

```
/*
 *   local_insertion_sort()
 *
 *   Do an insertion sort on a chain. Copies the list to an array,
 *   sorts that, then copies the array back into a list.
 */
static pointer
local_insertion_sort( pointer chain, size_t length )
{
     CLink    array[ARRAY_SORT_LIMIT];   /* the array of items */

     CLink    item;          /* an item */
     CLink    *ptr;          /* pointer for looping through the array */
     ushort   index;         /* a counter */


     /*
      *   Copy the chain into the array.
      */
     ptr = array;
     while ( chain != NULL )
     {
          *ptr++ = chain;
          chain = NEXT( chain );
     }

     /*
      *   Now sort the array. This is a standard insertion sort: the
outer
      *   loop looks at each item of the array, starting from the
_second_
      *   one. The inner loop scans from this pivot item down, shifting
      *   items upwards until it finds the spot to put the pivot item
in.
      */
     for ( index = 1; index < length; index ++ )
     {
          ptr = array + index;
          item = *ptr;               /* remember the item */
          ptr--;

          if ( *ptr > item )
          {
               while ( ptr >= array && *ptr > item )
               {
                    *(ptr+1) = *ptr;  /* scan downwards */
                    ptr--;
               }
               *(ptr+1) = item;      /* write the item */
          }
     }

     /*
      *   Now connect the items back into a chain again. The array is
      *   scanned backwards, to build the chain by putting items on the
      *   front.
      */
     ptr = array+length-1;
     chain = *ptr--;
     NEXT( chain ) = NULL; /* NB terminate _last_ item */
     for ( index = length-1; index > 0; index-- )
     {
          chain = CHN_CONS( chain, *ptr );
          ptr--;
     }

     return chain;
```

```
}

/* End chain.c */
```