Click to verify

This chapter provides advice on how best to use the C language when writing GNU software. Let’s dive into C best practices and coding style guidelines in detail, along with code examples and explanations:Use consistent indentation (usually 4 spaces or tabs) to improve code readability.Place opening braces { on the same line as the function or control statement, and closing braces } on a new line.Example: void print_numbers(int a, int b) { for (int i = a; i 0) /* OK, length is treated as counter variable containing multi values, not only 0 or 1 */ if (length == 0) /* OK, length is treated as counter variable containing multi values, not only 0 or 1 */ if (is_ok) /* OK, variable is treated as boolean */ if (!is_ok) /* OK, -||- */ if (is_ok == 1) /* Wrong, never compare boolean variable against 1! */ if (is_ok == 0) /* Wrong, use ! for negative check */ Always use /* comment */ for comments, even for single-line comment Always include check for C++ with extern keyword in header file Every function MUST include doxygen-enabled comment, even if function is static Use English names/text for functions, variables, comments Use lowercase characters for variables Use underscore if variable contains multiple names, eg. force_redraw. Do not use forceRedraw Never cast function returning void *, eg. uint8_t* ptr = (uint8_t *)func_returning_void_ptr(); as void * is safely promoted to any other pointer type Use uint8_t* ptr = func_returning_void_ptr(); instead Always use < and > for C Standard Library include files, eg. #include Always use "" for custom libraries, eg. #include "my_library.h" When casting to pointer type, always align asterisk to type, eg. uint8_t* t = (uint8_t*)var_width_diff_type Always respect code style already used in project or library Comments starting with // are not allowed. Always use /* comment */, even for single-line comment //This is comment (wrong) /* This is comment (ok) */ For multi-line comments use space+asterisk for every line /* * This is multi-line comments, * written in 2 lines (ok) */ /** * Wrong, use double-asterisk only for doxygen documentation */ /* * Single line comment without space before asterisk (wrong) */ /* * Single line comment in multi-line configuration (wrong) */ /* Single line comment (ok) */ Use 12 indents (12 * 4 spaces) offset when commenting. If statement is larger than 12 indents, make comment 4-spaces aligned (examples below) to next available indent void my_func(void) { char a, b; a = call_func_returning_char_a(a); /* This is comment with 12*4 spaces indent from beginning of line */ b = call_func_returning_char_a_but_func_name_is_very_long(a); /* This is comment, aligned to 4-spaces indent */ } Every function which may have access from outside its module, MUST include function prototype (or declaration) Function name MUST be lowercase, optionally separated with underscore _ character /* OK */ void my_func(void); void myfunc(void); /* Wrong */ void MYFunc(void); void myFunc(); When function returns pointer, align asterisk to return type /* OK */ const char* my_func(void); my_struct_t* my_func(int32_t a, int32_t b); /* Wrong */ const char *my_func(void); my_struct_t * my_func(void); Align all function prototypes (with the same/similar functionality) for better readability /* OK, function names aligned */ void set(int32_t a); my_type_t get(void); my_ptr_t* get_ptr(void); /* Wrong */ void set(int32_t a); const char * get(void); Function implementation MUST include return type and optional other keywords in separate line /* OK */ int32_t foo(void) { return 0; } /* OK */ static const char* get_string(void) { return "Hello world!\r"; } /* Wrong */ int32_t foo(void) { return 0; } Make variable name all lowercase with optional underscore _ character /* OK */ int32_t a; int32_t my_var; int32_t myvar; /* Wrong */ int32_t A; int32_t myVar; int32_t MYVar; Group local variables together by type void foo(void) { int32_t a, b; /* OK */ char a; char b; /* Wrong, char type already exists */ } Do not declare variable after first executable statement void foo(void) { int32_t a; a = bar(); int32_t b; /* Wrong, there is already executable statement */ } You may declare new variables inside next indent level int32_t a, b; a = foo(); if (a) { int32_t c, d; /* OK, c and d are in if-statement scope */ c = foo(); int32_t e; /* Wrong, there was already executable statement inside block */ } Declare pointer variables with asterisk aligned to type /* OK */ char* a; /* Wrong */ char *a; char * a; When declaring multiple pointer variables, you may declare them with asterisk aligned to variable name Structure or enumeration name MUST be lowercase with optional underscore _ character between words Structure or enumeration may contain typedef keyword All structure members MUST be lowercase All enumeration members SHOULD be uppercase Structure/enumeration MUST follow doxygen documentation syntax When structure is declared, it may use one of 3 different options: When structure is declared with name only, it MUST not contain _t suffix after its name. struct struct_name { char* a; char b; }; When structure is declared with typedef only, it has to contain _t suffix after its name. typedef struct { char* a; char b; } struct_name_t; When structure is declared with name and typedef, it MUST NOT contain _t for basic name and it MUST contain _t suffix after its name for typedef part. typedef struct struct_name { /* No _t */ char* a; char b; char c; } struct_name_t; /* _t */ Examples of bad declarations and their suggested corrections /* a and b MUST be separated to 2 lines */ /* Name of structure with typedef MUST include _t suffix */ typedef struct { int32_t a, b; } a; /* Corrected version */ typedef struct { int32_t a; int32_t b; } a_t; /* Wrong name, it MUST not include _t suffix */ struct name_t { int32_t a; int32_t b; }; /* Wrong parameters, MUST be all uppercase */ typedef enum { MY_ENUM_TESTA, my_enum_testb, } my_enum_t; When initializing structure on declaration, use C99 initialization style /* OK */ a_t a = { .a = 4, .b = 5, }; /* Wrong */ a_t a = {1, 2}; When new typedef is introduced for function handles, use _fn suffix /* Function accepts 2 parameters and returns uint8_t */ /* Name of typedef has `_fn` suffix */ typedef uint8_t (*my_func_typedef_fn)(uint8_t p1, const char* p2); Every compound statement MUST include opening and closing curly bracket, even if it includes only 1 nested statement Every compound statement MUST include single indent; when nesting statements, include 1 indent size for each nest /* OK */ if (c) { do_a(); } else { do_b(); } /* Wrong */ if (c) do_a(); else do_b(); /* Wrong */ if (c) do_a(); else do_b(); In case of if or if-else-if statement, else MUST be in the same line as closing bracket of first statement /* OK */ if (a) { } else if (b) { } else { } /* Wrong */ if (a) { } else { } /* Wrong */ if (a) { } else { } In case of do-while statement, while part MUST be in the same line as closing bracket of do part /* OK */ do { int32_t a; a = do_a(); do_b(a); } while (check()); /* Wrong */ do { /* ... */ } while (check()); /* Wrong */ do { /* ... */ } while (check()); Indentation is REQUIRED for every opening bracket if (a) { do_a(); } else { do_b(); if (c) { do_c(); } } Compound statement MUST include curly brackets, even in the case of a single statement. Examples below show bad practices if (a) do_b(); else do_c(); if (a) do_a(); else do_b(); Empty while, do-while or for loops MUST include brackets /* OK */ while (is_register_bit_set()) {} /* Wrong */ while (is_register_bit_set()); while (is_register_bit_set()) { } while (is_register_bit_set()) { } If while (or for, do-while, etc) is empty (it can be the case in embedded programming), use empty single-line brackets /* Wait for bit to be set in embedded hardware unit */ volatile uint32_t* addr = HW_PERIPH_REGISTER_ADDR; /* Wait bit 13 to be ready */ while (*addr & (1 py = (4); /* ... to this. In this example this is not a problem. */ /* Consider this ugly code, however it is valid by C standard (not recommended) */ if (a) /* If a is true */ if (b) /* If b is true */ SET_POINT(&p, 3, 4);/* Set point to x = 3, y = 4 */ else SET_POINT(&p, 5, 6);/* Set point to x = 5, y = 6 */ /* Evaluates to code below. Do you see the problem? */ if (a) if (b) (&p)->px = (3); (&p)->py = (4); else (&p)->px = (5); (&p)->py = (6); /* Or if we rewrite it a little */ if (a) if (b) (&p)->px = (3); (&p)->py = (4); else (&p)->px = (5); (&p)->py = (6); /* * Ask yourself a question: To which `if` statement does the `else` keyword belong? * * Based on first part of code, answer is straight-forward. To inner `if` statement when we check `b` condition * Actual answer: Compilation error as `else` belongs nowhere */ /* Better and correct implementation of macro */ #define SET_POINT(p, x, y) do { (p)->px = (x); (p)->py = (y); } while (0) /* 2 statements. No semicolon after while loop */ /* Or even better */ #define SET_POINT(p, x, y) do { \ /* Backslash indicates statement continues in new line */ (p)->px = (x); \ (p)->py = (y); \ } while (0) /* 2 statements. No semicolon after while loop */ /* Now original code evaluates to */ if (a) if (b) do { (&p)->px = (3); (&p)->py = (4); } while (0); else do { (&p)->px = (5); (&p)->py = (6); } while (0); /* Every part of `if` or `else` contains only `1` inner statement (do-while), hence this is valid evaluation */ /* To make code perfect, use brackets for every if-ifelse-else statements */ if (a) { /* If a is true */ if (b) { /* If b is true */ SET_POINT(&p, 3, 4);/* Set point to x = 3, y = 4 */ } else { SET_POINT(&p, 5, 6);/* Set point to x = 5, y = 6 */ } } Avoid using #ifdef or #ifndef. Use defined() or !defined() instead #ifdef XYZ /* do something */ #endif /* XYZ */ Always document if/elif/else/endif statements /* OK */ #if defined(XYZ) /* Do if XYZ defined */ #else /* defined(XYZ) */ /* Do if XYZ not defined */ #endif /* !defined(XYZ) */ /* Wrong */ #if defined(XYZ) /* Do if XYZ defined */ #else /* Do if XYZ not defined */ #endif Do not indent sub statements inside #if statement /* OK */ #if defined(XYZ) #if defined(ABC) /* do when ABC defined */ #endif /* defined(ABC) */ #else /* defined(XYZ) */ /* Do when XYZ not defined */ #endif /* !defined(XYZ) */ /* Wrong */ #if defined(XYZ) #if defined(ABC) /* do when ABC defined */ #endif /* defined(ABC) */ #else /* defined(XYZ) */ /* Do when XYZ not defined */ #endif /* !defined(XYZ) */ Documented code allows doxygen to parse and generate html/pdf/latex output, thus it is very important to do it properly at an early stage of the project. Use doxygen-enabled documentation style for variables, functions and structures/enumerations Always use \ for doxygen, do not use @ Always use 5x4 spaces (5 tabs) offset from beginning of line for text /** * \brief Holds pointer to first entry in linked list * Beginning of this text is 5 tabs (20 spaces) from beginning of line */ static type_t* list; Every structure/enumeration member MUST include documentation Align start of comments between different structure members to the same column /** * \brief This is point struct * ote This structure is used to calculate all point * related stuff */ typedef struct { int32_t x; /*!< Point X coordinate */ int32_t y; /*!< Point Y coordinate */ int32_t size; /*!< Point size. Since comment is very big, you may go to next line */ } point_t; /** * \brief Point color enumeration */ typedef enum { COLOR_RED, /*!< Red color */ COLOR_GREEN, /*!< Green color */ COLOR_BLUE, /*!< Blue color */ } point_color_t; Documentation for functions MUST be written in function implementation (source file usually) Function MUST include brief and all parameters documentation Every parameter MUST be noted if it is in or out for input and output respectively Function MUST include return parameter if it returns something. This does not apply for void functions Function can include other doxygen keywords, such as note or warning Use colon : between parameter name and its description /** * \brief Sum `2` numbers * \param[in] a: First number * \param[in] b: Second number * \return Sum of input values */ int32_t sum(int32_t a, int32_t b) { return a + b; } /** * \brief Sum `2` numbers and write it to pointer * ote This function does not return value, it stores it to pointer instead * \param[in] a: First number * \param[in] b: Second number * \param[out] result: Output variable used to save result */ void void_sum(int32_t a, int32_t b, int32_t* result) { *result = a + b; } If function returns member of enumeration, use ref keyword to specify which one /** * \brief My enumeration */ typedef enum { MY_ERR, /*!< Error value */ MY_OK /*!< OK value */ } my_enum_t; /** * \brief Check some value * \return \ref MY_OK on success, member of \ref my_enum_t otherwise */ my_enum_t check_value(void) { return MY_OK; } Use notation (`NULL` => NULL) for constants or numbers /** * \brief Get data from input array * \param[in] in: Input data * \return Pointer to output data on success, `NULL` otherwise */ const void * get_data(const void* in) { return in; } Documentation for macros MUST include hideinitializer doxygen command /** * \brief Get minimal value between `x` and `y` * \param[in] x: First value * \param[in] y: Second value * \return Minimal value between `x` and `y` * \hideinitializer */ #define MIN(x, y) ((x) < (y) ? (x) : (y)) Leave single empty line at the end of file Every file MUST include doxygen annotation for file and brief description followed by empty line (when using doxygen) /** * \file template.h * \brief Template include file */ /* Here is empty line */ Every file (header or source) MUST include license (opening comment includes single asterisk as this MUST be ignored by doxygen) Use the same license as already used by project/library /** * \file template.h * \brief Template include file */ /* * Copyright (c) year FirstName LASTNAME * * Permission is hereby granted, free of charge, to any person * obtaining a copy of this software and associated documentation * files (the "Software"), to deal in the Software without restriction, * including without limitation the rights to use, copy, modify, merge, * publish, distribute, sublicense, and/or sell copies of the Software, * and to permit persons to whom the Software is furnished to do so, * subject to the following conditions: * * The above copyright notice and this permission notice shall be * included in all copies or substantial portions of the Software. * * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR * OTHER DEALINGS IN THE SOFTWARE. * * This file is part of library_name. * * Author: FirstName LASTNAME */ Header file MUST include guard #ifndef Header file MUST include C++ check Include external header files outside C++ check Include external header files with STL C files first followed by application custom files Header file MUST include only every other header file in order to compile correctly, but not more (.c should include the rest if REQUIRED) Header file MUST only expose module public variables/types/functions Use extern for global module variables in header file, define them in source file later /* file.h ... */ #ifndef ... extern int32_t my_variable; /* This is global variable declaration in header */ #endif /* file.c ... */ int32_t my_variable; /* Actually defined in source */ Never include .c files in another .c file .c file should first include corresponding .h file, later others, unless otherwise explicitly necessary Do not include module private declarations in header file Header file example (no license for sake of an example) /* License comes here */ #ifndef TEMPLATE_HDR_H #define TEMPLATE_HDR_H /* Include headers */ #ifdef __cplusplus extern "C" { #endif /* __cplusplus */ /* File content here */ #ifdef __cplusplus } #endif /* __cplusplus */ #endif /* TEMPLATE_HDR_H */ Repository comes with always-up-to-date .clang-format file, an input configuration for clang-format tool. It can be seamlessly integrated with most of latest techno IDEs, including VSCode. Formatting then happens on the spot on each file save. AStyle is a great piece of software that can help with formatting the code based on input configuration. This repository contains astyle-code-format.cfg file which can be used with AStyle software. astyle --options="astyle-code-format.cfg" "input_path/*.c,*.h" "input_path2/*.c,*.h" Artistic style configuration is obsolete and no longer updated Repository contains eclipse-ext-kr-format.xml file that can be used with eclipse-based toolchains to set formatter options. It is based on K&R formatter with modifications to respect above rules. You can import it within eclipse settings, Preferences -> LANGUAGE -> Code Style -> Formatter tab. The home of Standard C++ on the web — news, status and discussion about the C++ standard on all compilers and platforms. Introduction This document is a modified version of a document from a committee formed at AT&T's Indian Hill labs to establish a common set of coding standards and recommendations for the Indian Hill community. The scope of this work is C coding style. Good style should encourage consistent layout, improve portability, and reduce errors. This work does not cover functional organization, or general issues such as the use of gotos. We(The opinions in this document do not reflect the opinions of all authors. This is still an evolving document. Please send comments and suggestions to pardo@cs.washington.edu or {rutgers,cornell,ucsd,ubc-cs,tektronix}!uw-beaver!june!pardo) have tried to combine previous work [1,6,8] on C style into a uniform set of standards that should be appropriate for any project using C, although parts are biased towards particular systems. Of necessity, these standards cannot cover all situations. Experience and informed judgement count for much. Programmers who encounter unusual situations should consult either experienced C programmers or code written by experienced C programmers (preferably following these rules). The standards in this document are not of themselves required, but individual institutions or groups may adopt part or all of them as a part of program acceptance. It is therefore likely that others at your institution will code in a similar style. Ultimately, the goal of these standards is to increase portability, reduce maintenance, and above all improve clarity. Many of the style choices here are somewhat arbitrary. Mixed coding style is harder to maintain than bad coding style. When changing existing code it is better to conform to the style (indentation, spacing, commenting, naming conventions) of the existing code than it is to blindly follow this document. "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 79 columns are not handled well by all terminals and should be avoided if possible. Excessively long lines which result from deep indenting are often a symptom of poorly-organized code. File Naming Conventions File names are made up of a base name, and an optional period and suffix. The first character of the name should be a letter and all characters (except the period) should be lower-case letters and numbers. The base name should be eight or fewer characters and the suffix should be three or fewer characters (four, if you include the period). These rules apply to both program files and default files used and produced by the program (e.g., "rogue.sav"). Some compilers and tools require certain suffix conventions for names of files [5]. The following suffixes are required: C source file names must end in .c Assembler source file names must end in .s The following conventions are universally followed: Relocatable object file names end in .o Include header file names end in .h. An alternate convention that may be preferable in multi-language environments is to suffix both the language type and .h (e.g. foo.c.h; or foo.ch). Yacc source file names end in .y Lex source file names end in .l C++ has compiler-dependent suffix conventions, including .c, ..c, .cc, .c.c, and .C. Since much C code is also C++ code, there is no clear solution here. In addition, it is conventional to use Makefile (not makefile) for the control file for make (for systems that support it) and "README" for a summary of the contents of the directory or directory tree. Program Files The suggested order of sections for a program file is as follows 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. 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. 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. 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. The functions come last, and should be in some sort of meaningful order. Like functions should appear together. A "breadth-first" approach (functions on a similar level of abstraction together) is preferred over depth-first (functions defined as soon as possible before or after their calls). Considerable judgement is called for here. If defining large numbers of essentially-independent utility functions, consider alphabetical order. 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 construction for getting them from a standard place, or define them relative to the current directory. The "include-path" option of the C compiler (-I on many systems) is the best way to handle extensive private libraries of header files; it permits reorganizing the directory structure without having to alter source files. Header files that declare functions or external variables should be included in the file that defines the function or variable. That way, the compiler can do type checking and the external declaration will always agree with the definition. Defining variables in a header file is often a poor idea. Frequently it is a symptom of poor partitioning of code between files. Also, some objects like typedefs and initialized data definitions cannot be seen twice by the compiler in one compilation. On some systems, repeating uninitialized declarations without the extern keyword also causes problems. Repeated declarations can happen if include files are nested and will cause the compilation to fail. Header files should not be nested. The prologue for a header file should, therefore, describe what other headers need to be #included for the header to be functional. In extreme cases, where a large number of header files are to be included in several different source files, it is acceptable to put all common #includes in one include file. It is common to put the following into each .h file to prevent accidental double-inclusion. #ifndef EXAMPLE_H #define EXAMPLE_H ... /* body of example.h file */ #endif /* EXAMPLE_H */ This double-inclusion mechanism should not be relied upon, particularly to perform nested includes. 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 with the opening /* in columns 1-2, a * in column 2 before each line of comment text, and the closing */ in columns 2-3. An alternative is to have ** in columns 1-2, and put the closing */ also in 1-2. /* * Here is a block comment. * The comment text should be tabbed or spaced over uniformly. * The opening slash-star and closing star-slash are alone on a line. */ /* ** Alternate format for block comments */ Note that grep '^.*' will catch all block comments in the file. Some automated program-analysis packages use different characters before comment lines as a marker for lines with specific items of information. In particular, a line with a '-' in a comment preceding a function is sometimes assumed to be a one-line summary of the function's purpose. 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 same line as the structure tag, and the closing brace (}) should be in column 1. struct boat { int wllength; /* water line length in meters */ int type; /* see below */ long sailarea; /* sail area in square mm */ }; /* defines for boat.type */ #define KETCH (1) #define YAWL (2) #define SLOOP (3) #define SQRIG (4) #define MOTOR (5) These defines are sometimes put right after the declaration of type, within the struct declaration, with enough tabs after the '#' to indent define one level more than the structure member declarations. When the actual values are unimportant, the enum facility is better. enums might be better anyway. enum bt { KETCH=1, YAWL, SLOOP, SQRIG, MOTOR }; struct boat { int wllength; /* water line length in meters */ enum bt type; /* what kind of boat */ long sailarea; /* sail area in square mm */ }; Any variable whose initial value is important should be explicitly initialized, or at the very least should be commented to indicate that C's default initialization to zero is being relied upon. The empty initializer, "{}", should never be used. Structure initializations should be fully parenthesized with braces. Constants used to initialize longs should be explicitly long. Use capital letters; for example two long "2l" looks a lot like "21", the number twenty-one. int x = 1; char *msg = "message"; struct boat winner[] = { { 40, YAWL, 6000000L }, { 28, MOTOR, 0L }, { 0 }, }; In any file which is part of a larger whole rather than a self-contained program, maximum use should be made of the static keyword to make functions and variables local to single files. Variables in particular should be accessible from other files only when there is a clear need that cannot be filled in another way. Such usage should be commented to make it clear that another file's variables are being used; the comment should name the other file. If your debugger hides static objects you need to see during debugging, declare them as STATIC and #define STATIC as needed. The most important types should be highlighted by typedeffing them, even if they are only integers, as the unique name makes the program easier to read (as long as there are only a few things typedeffed to integers!). Structures may be typedeffed when they are declared. Give the struct and the typedef the same name. typedef struct splodge_t { int sp_count; char *sp_name, *sp_alias; } splodge_t; The return type of functions should always be declared. If function prototypes are available, use them. One common mistake is to omit the declaration of external math functions that return double. The compiler then assumes that the return value is an integer and the bits are dutifully converted into a (meaningless) floating point value. "C takes the point of view that the programmer is always right." -- Michael DeCorte" 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 stop4. "Tabstops" can be blanks (spaces) inserted by your editor in clumps of 2, 4, or 8. Use actual tabs where possible. Do not default to int; if the function does not return a value then it should be given return type void. #define void or #define void int for compilers without the void keyword. If the value returned requires a long explanation, it should be given in the prologue; otherwise it can be on the same line as the return type, tabbed over. The function name (and the formal parameter list) should be alone on a line, in column 1. Destination (return value) parameters should generally be first (on the left). All formal parameter declarations, local declarations and code within the function body should be tabbed over one stop. The opening brace of the function body should be alone on a line beginning in column 1. Each parameter should be declared (do not default to int). In general the role of each variable in the function should be described. This may either be done in the function comment or, if each declaration is on its own line, in a comment on that line. Loop counters called "i", string pointers called "s", and integral types called "c" and used for characters are typically excluded. If a group of functions all have a like parameter or local variable, it helps to call the repeated variable by the same name in all functions. (Conversely, avoid using the same name for different purposes in related functions.) Like parameters should also appear in the same place in the various argument lists. Comments for parameters and local variables should be tabbed so that they line up underneath each other. Local variable declarations should be separated from the function's statements by a blank line. Be careful when you use or declare functions that take a variable number of arguments ("varargs"). There is no truly portable way to do varargs in C. Better to design an interface that uses a fixed number of arguments. If you must have varargs, use the library macros for declaring functions with variant argument lists. If the function uses any external variables (or functions) that are not declared globally in the file, these should have their own declarations in the function body using the extern keyword. Avoid local declarations that override declarations at higher levels. In particular, local variables should not be redeclared in nested blocks. Although this is valid C, the potential confusion is enough that lint will complain about it when given the -h option. Whitespace int i;main(){for(;i["]next = bar = tmp; In the second example, the address of "bar->next" may be computed before the value is assigned to "bar". bar = bar->next = tmp; In the third example, bar can be assigned before bar->next. Although this appears to violate the rule that "assignment proceeds right-to-left", it is a legal interpretation. Consider the following example: long i; short a[N]; i = old i = a[i] = new; The value that "i" is assigned must be a value that is typed as if assignment proceeded right-to-left. However, "i" may be assigned the value "(long)(short)new" before "a[i]" is assigned to. Compilers do differ. Be suspicious of numeric values appearing in the code ("magic numbers"). Avoid preprocessor tricks. Tricks such as using /**/ for token pasting and macros that rely on argument string expansion will break reliably. #define FOO(string) (printf("string = %s",(string))) ... FOO(filename); Will only sometimes be expanded to (printf("filename = %s", (filename))) Be aware, however, that tricky preprocessors may cause macros to break accidentally on some machines. Consider the following two versions of a macro. #define LOOKUP(chr) (a['c'+(chr)]) /* Works as intended. */ #define LOOKUP(c) (a['c'+(c)]) /* Sometimes breaks. */ The second version of LOOKUP can be expanded in two different ways and will cause code to break mysteriously. Become familiar with existing library functions and defines. (But not too familiar. The internal details of library facilities, as opposed to their external interfaces, are subject to change without warning. They are also often quite unportable.) You should not be writing your own string compare routine, terminal control routines, or making your own defines for system structures. "Rolling your own" wastes your time and makes your code less readable, because another reader has to figure out whether you're doing something special in that reimplemented stuff to justify its existence. It also prevents your program from taking advantage of any microcode assists or other means of improving performance of system routines. Furthermore, it's a fruitful source of bugs. If possible, be aware of the differences between the common libraries (such as ANSI, POSIX, and so on). Use lint when it is available. It is a valuable tool for finding machine-dependent constructs as well as other inconsistencies or program bugs that pass the compiler. If your compiler has switches to turn on warnings, use them. Suspect labels inside blocks with the associated switch or goto outside the block. Wherever the type is in doubt, parameters should be cast to the appropriate type. Always cast NULL when it appears in non-prototyped function calls. Do not use function calls as a place to do type cheating. C has confusing promotion rules, so be careful. For example, if a function expects a 32-bit long and it is passed a 16-bit int the stack can get misaligned, the value can get promoted wrong, etc. Use explicit casts when doing arithmetic that mixes signed and unsigned values. The inter-procedural goto, longjmp, should be used with caution. Many implementations "forget" to restore values in registers. Declare critical values as volatile if you can or comment them as VOLATILE. Some linkers convert names to lower-case and some only recognize the first six letters as unique. Programs may break quietly on these systems. Beware of compiler extensions. If used, document and consider them as machine dependencies. A program cannot generally execute code in the data segment or write into the code segment. Even when it can, there is no guarantee that it can do so reliably. ANSI C Modern C compilers support some or all of the ANSI proposed standard C. Whenever possible, write code to run under standard C, and use features such as function prototypes, constant storage, and volatile storage. Standard C improves program performance by giving better information to optimizers. Standard C improves portability by insuring that all compilers accept the same input language and by providing mechanisms that try to hide machine dependencies or emit warnings about code that may be machine-dependent. Compatibility Write code that is easy to port to older compilers. For instance, conditionally #define new (standard) keywords such as const and volatile in a global .h file. Standard compilers pre-define the preprocessor symbol __STDC__8. Some compilers predefine __STDC__ to be 0, in an attempt to indicate partial compliance with the ANSI C standard. Unfortunately, it is not possible to determine which ANSI facilities are provided. Thus, such compilers are broken. See the rule about "don't write around a broken compiler unless you are forced to." The void* type is hard to get right simply, since some older compilers understand void but not void*. It is easiest to create a new (machine- and compiler-dependent) VOIDP type, usually char* on older compilers. #if __STDC__ typedef void *voidp; # define COMPILER_SELECTED #endif #ifdef A_TARGET # define const # define volatile # define void int typedef char *voidp; # define COMPILER_SELECTED #endif #ifdef ... ... #endif #ifdef COMPILER_SELECTED # undef COMPILER_SELECTED #else { NO TARGET SELECTED! } #endif Note that under ANSI C, the '#' for a preprocessor directive must be the first non-whitespace character on a line. Under older compilers it must be the first character on the line. When a static function has a forward declaration, the forward declaration must include the storage class. For older compilers, the class must be "extern". For ANSI compilers, the class must be "static". but global functions must still be declared as "extern". Thus, forward declarations of static functions should use a #define such as FWD_STATIC that is #ifdeffed as appropriate. An "#ifdef NAME" should end with either "#endif" or "#endif /* NAME */", not with "#endif NAME". The comment should not be used on short #ifdefs, as it is clear from the code. ANSI trigraphs may cause programs with strings containing "??" may break mysteriously. Formatting The style for ANSI C is the same as for regular C, with two notable exceptions: storage qualifiers and parameter lists. Because const and volatile have strange binding rules, each const or volatile object should have a separate declaration. int const *s; /* YES */ int const *s, *t; /* NO */ Prototyped functions merge parameter declaration and definition in to one list. parameters should be commented in the function comment. /* * 'bp': boat trying to get in. * 'stall': a list of stalls, never NULL. * returns stall number, 0 => no room. */ int enter_pier (boat_t const *bp, stall_t *stall) { ... 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 Such automatic type promotion is called widening. For older compilers, the widening rules require that all char and short parameters are passed as ints and that float parameters are passed as doubles. 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 compilers10. Note that using PROTO violates the rule "don't change the syntax via macro substitution." It is regrettable that there isn't a better solution. #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 =, never use an exact comparison (== or !=). Compilers have bugs. Common trouble spots include structure assignment and bitfields. You cannot generally predict which bugs a compiler has. You could write a program that avoids all constructs that are known broken on all compilers. You won't be able to write anything useful, you might still encounter bugs, and the compiler might get fixed in the meanwhile. Thus, you should write "around" compiler bugs only when you are forced to use a particular buggy compiler. Do not rely on automatic beautifiers. The main person who benefits from good program style is the programmer him/herself, and especially in the early design of handwritten algorithms or pseudo-code. Automatic beautifiers can only be applied to complete, syntactically correct programs and hence are not available when the need for attention to white space and indentation is greatest. Programmers can do a better job of making clear the complete visual layout of a function or file, with the normal attention to detail of a careful programmer. (In other words, some of the visual layout is dictated by intent rather than syntax and beautifiers cannot read minds.) Sloppy programmers should learn to be careful programmers instead of relying on a beautifier to make their code readable. Accidental omission of the second "=" of the logical compare is a problem. Use explicit tests. Avoid assignment with implicit test. abool = bbool; if (abool) { ... When embedded assignment is used, make the test explicit so that it doesn't get "fixed" later. while ((abool = bbool) != FALSE) { ... while (abool = bbool) { ... /* VALUSED */ while (abool = bbool, abool) { ... Explicitly comment variables that are changed out of the normal control flow, or other code that is likely to break during maintenance. Modern compilers will put variables in registers automatically. Use the register sparingly to indicate the variables that you think are most critical. In extreme cases, mark the 2-4 most critical values as register and mark the rest as REGISTER. The latter can be #