

Nectarine City

HANDBOOK OF
C PROGRAMMING STYLE

2010 EDITION

Copyright ©2008, 2009, 2010 Nectarine City LLC. This book is licensed under the “Creative Commons Attribution Share-Alike” license. To view a copy of this license, either visit the URL

<http://creativecommons.org/licenses/by-sa/3.0>

or, send a letter to

Creative Commons,
171 2nd Street, Suite 300,
San Francisco, California,
94105, USA.

Nectarine City LLC is available to conduct full and half-day workshops based on the contents of this manual, as well as other topics of interest to programmers. For details, please contact us at hello@nectarine-city.com, or visit our web site at <http://www.nectarine-city.com>.

1 – Understand The Need For Style

It is possible to write a correct, efficient program that still has big problems. Consider the following C program.

```
main(int v,char**c){
    if(--v){
        main(v,c); c[v][0]^=0x80;
        for(c[0][0]=0;c[+c[0][0]]!=0;)
            if(c[c[0][0]][0]>0)
                puts(c[c[0][0]]);
        puts("-- ");
        main(v,c);}}
```

This program prints all proper subsets of its command line arguments. It would take the next programmer an hour to figure out what this program did, given just the above source code. An ideal program would have an obvious meaning to any programmer immediately as it is read.

In an effort to curb programs like the one above, programmers develop rules of style. There are several kinds of such rules. *Prohibitions* prohibit certain language features or uses. For example, a rule that prohibits the use of `setjmp` and `longjmp` from the standard library. *Best Practices* are broad based suggestions about how to stay out of trouble. For example, advice to not write self-evident comments. *Aesthetic rules* make code easier on the eyes. For example, a rule making an underscore, not a change in letter case, the way to separate words in an identifier. *Uniformity rules* make code all look the same. For example, a rule that says a left curly brace never goes on a line of their own, unless it is the first curly brace in a function definition. *Idioms* codify the usual way to express common things, or to remedy typical problems. For example, a rule that says constants or other values that are not legal lvalues should be put on the left of a `==` operator. This handbook values rules of style in that order. Outright prohibitions begin with the phrase “Do not.” Everything else is more or less advisory.

1.1 – Be Pragmatic

Pragmatics is a technical term from Linguistics. We rely on a subtle cooperative spirit in communication in order to make sense of certain sentences. Consider the following English sentence.

Ben told me that Jack went to see Star Wars. He said he hated it.

You probably made a large number of assumptions when reading this. For example, you probably assumed that “Star Wars” should be understood as the name of a movie, that the first “he” refers to Ben, and that the second “he” refers to Jack. You can perform these disambiguations effortlessly. This seems to be, at least partially, because our minds are naturally tuned to assume that the speaker is not purposefully trying to be deceptive or confusing, and that you know a lot of the same stuff the speaker does. Pragmatics is the study of these kinds of linguistic phenomena.

In natural language we must rely on pragmatics to communicate at all. Formal languages, programming languages included, convey meaning without relying upon pragmatics. No spirit of cooperation is needed to for a program to run, just a compiler. Good programming style is essentially the consideration of pragmatics while programming. Pragmatics does not make your program immediately more correct, or faster. Pragmatics instead makes your program easier to read and understand, and therefore more correct and fast in the long term.

1.2 – Beware False Pragmatics

Some decisions appear to be pragmatic, but are not when put into proper perspective. For example, one very famous programmer insists on writing all of his external function declarations in the following way.

```
extern Foo_Bar corge_gralt P_ ((int, int));
```

This allows the use of a preprocessor macro `P_` to eliminate the arguments from the function prototype if we’re ever trapped in a time warp, forcing us to use C Compilers from before C was standardized. The same programmer would write the body of the above function as follows.

```
Foo_Bar  
corge_gralt()  
    int x;  
    int y;  
{  
    return x + y;  
}
```

The programmer has what seem to be pragmatic reasons for doing so. When asked about it on a project mailing list, he replied as follows.

The reason I prefer K&R style in function definitions is that the argument type declarations are easier to read when not inside the parentheses.

The desire to make the code easier to read is certainly a desire to be pragmatic. It isn't actually pragmatic, however. When C programmers see the old K&R style function definition, a little alarm goes off in their head. Is this very old code, or some kind of joke? Does one of the other programmers not know what they're doing? Writing function definitions this way actually makes the code harder to read, for those reasons.

There is a similar false pragmatism at work in the above definition of `corge_gralt`. Notice that the return type is on a line of its own. The reason for this is described in the following passage taken from the style manual for the project.

It is also important for function definitions to start the name of the function in column one. This helps people to search for function definitions, and may also help certain tools recognize them.

This too appears to be a pragmatic decision, as it makes searching easier for people and tools. If the code is written this way, one could find the definition of `corge_gralt` by using the shell command `grep ^corge_gralt *.c`. But, such a line break looks so alien to most C programmers that it too raises more questions and anxiety than it does promote clarity. Also, tools have no problem finding the start of function definitions regardless of where the lines break. Only in some cases of using macros to define functions do the tools begin to fail, and in such cases breaking the line this way does not help. Even if the name of the function being defined does not start in column one, one can as easily run the command `etags -I *.c`. The output of this command can be used by other tools automatically, without re-input of `grep` results by the programmer.

2 – Write Code For People

Rules of style are largely guides which lead programmers to more strongly consider the concerns of later programmers. It is possible to interpret such rules as purely self-interested, as the later programmer usually turns out to be the same as the original programmer, only after enough time has passed for him or her to forget any relevant initial premises.

2.1 – Name Things Appropriately

Most of what gives a program good style is the quality of the names for its variables, functions, macros, files, and other entities. Excellent names have the following traits: they can be read aloud with unambiguous pronunciation, they are syntactically and lexically graceful when manipulated by an editor or debugger, they have obvious and/or intuitive meaning, they are unsurprising, they clearly distinguish the thing they name from other similar things, and they are not easily confused with any other names.

Avoid abbreviation. `i` is not an abbreviation when used as an index variable in a loop. `rbuf` is an abbreviation, and not a good name for a `receive_buffer`. This is because when the reader sees `rbuf` they must think “arr buf means receive buffer” every time they see it, until it fades into becoming another piece of obscure jargon. Do not fear long names. Any decent editor will make using long names easy.

If a name has parts, separate the parts with an underscore. Do not use mixed case to indicate word breaks. Put general components of a name to the left, specific components to the right. Name things in a way which is conducive to smart use of a completion feature in the program editor.

Use names in all upper case for C Preprocessor names. Use upper case in integer literals, as in `0x1F2EL`, otherwise readers may confuse `1` with `l`, the later of which is a legal character in C integer literals. Use all lower case for everything else.

Use `typedef`, and not the C Preprocessor, to name types. Names defined with `typedef` make it to the debugger, C preprocessor names do not. Also, it is awkward to fit large structures spanning several lines into the body of a `#define`.

A good name has three parts, the beginning and ending parts being optional. First, if the name is in a module with very generic names (for example functions named `print` and `delete`), the name may begin with a single word, the shorter the better, ideally related to the main data structure used by the module (for example `stack_print` and

`stack_delete`). This is a kind of phony namespace, and should be avoided if possible. The flat namespace is a deficiency in the language that must in many cases be worked around. The name for a phony namespace needs to be different than everything else, but of course what *everything else* might be can not be predicted. It's useless to go from `pop` to `stack_pop` if that's what the other modules do too. Nevertheless, there is always need to divide names into phony namespaces in large C programs. Do so with a short meaningful prefix and hope for the best.

Second is the name itself, which should be a verb, noun, or predicate followed by some adverbs or adjectives. Function names should be verbs or verb phrases that begin with a verb, as in `melt_quickly` (not `quickly_melt`). Everything other than function names should be nouns or noun phrases that begin with a noun, as in `circle_red` (not `red_circle`). Putting the noun first makes the relationship between names far more clear, and is less of an annoyance than that caused by the fact that “circle red” is not grammatical in English. If there is no similar name and the adjective is included as part of a common English idiom, use the natural order of words in the idiom, as in `dental_floss` rather than `floss_dental`. That is, consider idiomatic noun phrases as composite nouns. Names for predicates nearly always begin with `is_`, as in `is_round`.

Finally, a name may end in `_s`, `_e`, or `_t`. `_s` and `_e` disambiguate `struct` and `enum` names (respectively) from `typedef` names. `_t` makes type names consistent with type names from the standard library, and is also useful when you expect to name variables after the type (as in declaring a variable named `list` with the type `list_t`). These suffixes are entirely optional, as in the following example.

```
typedef int node_data_t;
typedef enum mode_e { butter, mayonaise } mode_t;
typedef struct node_s {
    mode_t mode;
    node_data_t data;
    struct node_s *next;
} node;
```

Do not put other type indicators into a name. Do not use “Hungarian Notation”, such as beginning the names of variables of type `const int` with a lowercase `ci`. These are hacks to work around deficiencies in bad debuggers and worse linkers. Find better tools.

Use names to make the meaning of literal data values explicit. For example, write `PI / 2.0` rather than `1.57079633`.

Name source files as any other C identifier (avoid abbreviation, separate parts with an underscore, etc.). Name C source files with the suffix `.c`. Name header files with the suffix `.h`. Do not use upper case in source file names, and avoid upper case in names of non-source files associated with the project.

2.2 – Use Whitespace Consistently

Do not use tabs; use only spaces. It is impossible to predict how a tab will be displayed.

A single level of indentation is two spaces. A popular free software project has the following in its style guide.

There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

In short, 8-char indents make things easier to read, and have the added benefit of warning you when you're nesting your functions too deep. Heed that warning.

If you have been looking at your screen for 20 straight hours, you're screwed anyway, and should get some rest. The code which the above style guide is talking about has more than three levels of indentation all over the place. Eight or four space indentation is too ragged of a left line for the eyes to easily move down while reading code. Use two spaces.

In expressions, there should be no whitespace between the following lexical elements; function calls, as in `f(x)`; pointer dereference operators, as in `*p` and `&a`; field selection operators, as in `p->next` and `s.foo`; increment and decrement operators, as in `i++`; the array index operator, as in `a[3]`, and constant pointer declarations, as in `*const`. Especially, always put a space after the right parenthesis of a cast, unless the cast is

an argument in a preprocessor macro invocation. Anything other than space, punctuation, or another right parenthesis after a right parenthesis should look weird. Put space after, but not before, commas. Put space around all assignment, comparison, arithmetic, and logical operators.

It is generally more aesthetically pleasing to see reference and dereference operators separated from their argument by a space. This is especially true in the case of `*const` declarations, which must be separated from their argument by a space anyway. However, years of following the original documentation for the C language have made it so that most programmers are surprised to see code as in the following example. Spaces around the asterisk read as multiplication.

```
int * a = NULL;
int j = 3;
a = & j;
j = * a * 3;
```

When defining a function, put the first left curly brace on its own line. Otherwise put the left curly brace at the end of the last line of the head of the statement, as follows.

```
if (foo) {
    corge();
    gralt();
}
```

Put the last right curly brace on its own line, or followed by the `while` in a `do while` statement, a type name in a `typedef struct`

declaration, or a semicolon after a `struct` declaration, as follows.

```
do {
    gralt();
    corge();
} while (bar);

struct node_s {
    int data;
    struct node_s *other;
};

typedef struct {
    float length;
    float width;
    float height;
} zone_t;
```

Note that the head of the statement may span multiple lines, as in the following example.

```
for (i = 0, j = 0, x = 0.5, y = 0.5;
     i <= FOO && j < BAR && x * y < CORGE;
     i++, j++) {
    argle(x, y, gralt(i, j));
}
```

Especially do not omit whitespace between keywords and left parenthesis, as in `return(foo)`. Use the space before the opening parenthesis to distinguish function calls from special syntactic forms.

Always put a space after `//` when writing a comment, unless commenting out a single line of code, in which case do not put a space after `//`. Do not use `//` style comments to comment out multiple lines of code.

Do not “exdent” `case` and `goto` labels. In a `switch` statement, or a block with a label, indent labels and `case` stanzas one indent level (two spaces), and the rest of the code two indent levels (four spaces), as in

the following example.

```
switch (foo) {
  case pants:
    gralt:
      bar();
      break;
  case shirt:
    if (corge()) goto gralt;
    break;
  default:
    break;
}
```

Write one statement per line. If the intention is that no more code will ever be added to the body of a loop or conditional which is one short line of code, then omit any curly braces and, if there is room, put the body on the same line as the loop or conditional construction, as in the following good examples.

```
while (*p) p++;

if (foo) {
  bar();
} else if (corge) {
  gralt();
}
```

Use line breaks to keep each line shorter than 78 characters. This makes code fit the 80 character wide displays while giving the editor the leftmost and rightmost columns to use for special marks. The convention of 80 columns is not in support of ancient display hardware, but for the same reason as newspapers use 1.83 inch wide columns of text. Reading is difficult when the end of one line is too far away from the beginning of the next. It is not uncommon for programmers to edit code in several windows, each of which is narrow enough to display a separate file side by side. Assuming those windows are about 80 columns wide is useful and readable.

Indent the continuation of an expression over several lines by starting subsequent lines one indent level (two spaces) in from the first. Tend to

break lines just before names, as in the following example.

```

argle_bargle_squared = argabarg +
    argle_bargle_argle_x * argle_bargle_x +
    argle_bargle_y * argle_bargle_y;

```

Indent the continuation of the function argument list or contents of the predicate clause in a statement by aligning subsequent lines with the first parenthesis, as in the following good example.

```

if (is_this_long(imagine_it_is,
                foo + bar - baz * corge) &&
    is_that_long(not_so_much)) {
    gralt();
}

```

When a statement goes over multiple lines, indent all lines other than the first. Use extra indentation to line up function arguments and make the structure of expressions obvious. Break lines intelligently so that the arguments don't all get forced into a narrow column to the right. Consider the first bad example and the subsequent good example below.

```

the_results_are_in->sandwich_ham_mayo = foo_bar(a,
                                                belt,
                                                keeps,
                                                pants,
                                                up);

x = 5;

the_results_are_in->sandwich_ham_mayo =
    foo_bar(a, belt, keeps, pants, up);
x = 5;

```

Align tabular data with extra whitespace when it makes sense to do so, as in the following example, but use only spaces, never tabs. Use the regular expression alignment feature of Emacs for this, if possible.

```

int   gralt_x = 52;
int   gralt_y = 40;
float gralt_up = 39.0;

```

Do not use code analyzers to correct the indentation and line breaks of large bodies of source code. It is fine to use such analysis to find

problems and then fix them by hand, but the only thing performing programmatic manipulations of source code should be an interactive editor and the C Preprocessor.

2.3 – Use Assertions

Assertions are the first line of defense against code maintenance problems. There is no better mechanism for a programmer to make explicit their expectations about a certain position in a program. It is difficult to write too many assertions, so if in doubt, put the assertion in.

Assertions will not be executed in production builds, so do not rely on assertions to catch legitimate run-time errors. The following is not just bad style, but in error, for example.

```
x = malloc(sizeof(foo_t));
assert(x != NULL);
```

Avoid writing assertions that test several cases at the same time in one assertion statement, as in the following bad example.

```
assert(p != NULL && i < 5 && j > 17);
```

Instead, separate out each clause into its own assertion. This will pinpoint which part of the assertion has gone bad without having to run again under the debugger. For example, rewrite the above assertion as follows.

```
assert(p != NULL);
assert(i < 5);
assert(j > 17);
```

2.4 – Write Meaningful Comments

An incorrect comment is as serious a defect as incorrect program logic; it is a bug waiting to happen. Keep comments correct. Make sure that joke and non-sequitur comments can not be mistaken as serious.

Use `//` style comments, not `/* */` style comments. Do not use more than two slashes to indicate different kinds of comments, or to give the look of indentation. Indent comments to line up with the left edge of

the code they are commenting on, as in the following good example.

```
// "Corge" was the name of one of mg's cats.
void corge()
{
    // XXX this is a hack for until we get
    // the sound driver working
    printf("Meow I tell you!\n");
}
```

Put a short comment at the beginning of each file indicating the name of the file, and the license under which the file may be distributed. Do not put in revision control expansions such as `Log`, or instructions for the editor such as `-*- indent-width: 2 -*-`.

Write comments near the code being commented upon. Keep general comments at the start of the file.

Do not write self evident comments, as in the following bad example.

```
// set n to n divided by 3
n = n / 3;
```

Avoid comments that are better expressed as an assertion, as in the following bad example.

```
// will not work on NULL strings!
while (*p++ = *q++);
```

The above comment is better expressed as assertions, as follows.

```
assert(p != NULL);
assert(q != NULL);
while (*p++ = *q++);
```

To comment out a single line of code, use `//`, but do not put a space in between `//` and the first non-whitespace character of the line being commented out. Put the first character of `//` where the start of the commented out statement would be. To comment out multiple lines of code, use `#if 0` and `#endif`. Do not reinstate commented out sections of code by changing from `#if 0` to `#if 1`. If it is that important or interesting, name the condition of compilation explicitly, as in `#ifdef FOO`. Prefer to use whether a name is defined or not as the indicator for conditional compilation, rather than what the name is defined as,

unless the name has a natural intuitive value, such as a version number. Consider the following bad example.

```
#if F00 == 1
    corge();
#else
    gralt();
#endif
```

Presumably the above code is to conditionally compile over the F00 feature. The intention is clearer when the command to build the code simply defines F00. If it has to set F00 to 1, it is unclear whether some other value might be meaningful. Make the semantics of conditional compilation defines explicit; use definition or lack of definition for the inclusion or lack of inclusion of a particular feature, respectively. Use numbers for everything else.

Conditional compilation is configured on a syntactically hostile command line, so it is best to only require that such a command line either define a macro or not. Selecting a value for a preprocessor definition is too much to ask of a command line.

Avoid putting any structured data inside of comments. Do not use documentation systems that require comments to be formatted in specific ways. Do not use comment templates, as in the following bad example taken from real production code. They're nearly always useless noise.

```
/******
 * Function Name : ADC12_Init
 * Description   : This routine is used to initialize
 *                the ADC12 registers to their reset
 *                values
 * Input        : None.
 * Return       : None.
 *****/
void ADC12_Init(void)
{
    /* Initiate ADC12 registers to their reset values */
    ADC12->CSR = 0x0000;
    ADC12->CPR = 0x0005;
}
```

There are many glaring problems with the above code. For example, “Return: None” is just restating the declaration of the void return

type. But the real problem is the use of templates in general. In theory, one could programmatically parse comments like the template above, and use the results to get printable documentation. It would then seem to make sense to demand inclusion of templated fields in a comment above every function, such as the “Return: None.”, even though it was also just restating the code itself, as it would give a uniform stream of data to this documentation system. It is certainly the case that keeping code and documentation coherent is a big problem. Cluttering up the code with template comments isn’t the solution. Neither is *any* kind of structured data in comments. Documentation that contains the information in the template above is no more useful than the code itself. What the reader really needs to know is what an ADC12 register is, what makes it different from some other kind of register, what “reset” means in context, and where one can get the data sheets for the devices in question. Documentation for the entry points into a library must present a high level view of how the library can be used, not just a concatenation of per-function documentation strings. Write such documentation in a separate file, and at most write how to find that file as a comment in the code.

Do not put names, initials, or date information in comments. Use revision control, not comments, to store the answers to the questions like “who wrote this?” and “when was it written?” Everything else that can be said about good comments can also be said about good annotations in a revision control system.

2.5 – Avoid Tricky Expressions

Avoid dereference-and-increment idioms, as in `a = *p++`. These are almost always used incorrectly, and are never any faster than performing the dereference and the increment in separate statements. Likewise, avoid assignment in predicate expressions. Consider the following statement.

```
while (*p++ = *q++);
```

Most C programmers will immediately identify this as copying a zero-terminated array (probably an ASCII NUL terminated string), but will encounter a fencepost error if they attempt to use either pointer after the statement. After the above statement runs, `p` and `q` point to addresses which are almost certainly incorrect to dereference. The following better code fragment leaves `p` and `q` pointing at the zero terminator, and moves

the assignment out of the predicate for the `while` loop.

```
while (*q) {
    *p = *q;
    p++;
    q++;
}
```

Some compilers will warn about assignment in a predicate context, demanding that extra parenthesis be placed around the assignment to eliminate the complaint. Do not use this misfeature. Instead, move the assignment out of the predicate.

Use extra parenthesis to make the meaning of complex expressions explicit. Especially clarify expressions involving `&`, `!`, `<<`, `>>`, `^`, and `?:` operators, which have counter-intuitive precedence and associativity. Conversely, do not over parenthesize obvious or trivial expressions.

Do not put expressions with side effects in the argument lists of function calls, as in `f(a++, a++)`. The semantics are too obscure and mind-bending.

2.6 – Declare and Initialize Locals First

Initialize variables in their declarations. It is perhaps easier for a compiler to optimize out an extra variable initialization than it is to detect use of an uninitialized variable.

Do not declare local variables in `for` expressions. Consider the following pathological example.

```
int j = 30;
for (int i = 0, j = 17; i < j; i++, j--)
    printf("%d %d\n", i, j);
printf("%d\n", j);
```

It is unclear, even to some compiler authors, what the precise scope rules for the variable `j` are in the above code. Are there two `j` variables or one? If there are two, which `j` is being compared to `i`, and which is decremented? How can one initialize the outermost declaration of `j` while declaring `i`? What is the value of `j` for the final call to `printf`? The following code shows an equivalent program; write it this

way instead.

```
int i = 0;
int j = 30;
int k = 17;
for (assert(i == 0 && k == 17); i < k; i++, k--)
    printf("%d %d\n", i, k);
printf("%d\n", j);
```

Give each local variable its own type declaration. Code such as the following example is confusing.

```
int *p, q = 0;
```

Is `q` an `int` or an `int *`? Are both of these variables meant to be initialized to zero, or just `q`? Write out the full type for every variable, making the answers to those questions completely explicit, as follows.

```
int *p = NULL;
int q = 0;
```

Declare variables at the beginning of their block, not among the other statements, as in the following bad example.

```
int i = 0;
for (assert(i == 0); i < 7; i++) foo(i);
int j = 0;
for (assert(j == 0); j < 13; j++) bar(j);
baz (i, j);
```

2.7 – Use `const` And `volatile` Correctly

Use `const` to declare that a variable may only be read, and never written to. Always write `const` as far to the left as possible, as in the following examples.

```
const int a;
const int *b;
const uint64_t *const c;
const int *const *const d;
```

It is more consistent to write `int const` than `const int`, since one *must* write `*const` when declaring a constant pointer. However, declarations of the form `int const a;` are seldom used, and look strange to most experienced C programmers.

Do not cast away `const`, or use other tricks to write to values declared `const`, as in the following bad example.

```
void foo(const int *p)
{
    *((int *) p) = 3;
}

void bar(void)
{
    const int i = 7;
    foo(&i);
}
```

`foo` is only legal to call on a pointer to non-`const` data, but the compiler can't always ascertain this at compile time. For the above code, compilers are free to generate programs that crash or otherwise fail at run time, although few do. Declaring a variable `const` has little impact on the compiler output, but adds a lot of information for other programmers, especially to function arguments.

On the other hand, `volatile` has a great deal of impact on compiler output. `volatile` should only be used in code for embedded systems and device drivers, where reading and writing specific memory locations yields up specific hardware related results. Do not make any other use of `volatile`. Like `const`, write `volatile` as far to the left as possible. If code is only going to read a `volatile` variable, the associated variable should be declared `const volatile`, not `volatile const`.

3 – Cooperate With The Compiler

A lot of ugly code is written in the name of optimization. The maxim “premature optimization is the root of all evil” is especially true of the evils of bad programming style. Write programs first to be correct, then to be clear and readable, and then fast, in that order, without skipping steps. Do not try to do better than the compiler optimizer until there is strong evidence (profiler output from runs on real data, for example) that it is both failing to optimize, and that the code in question is truly a bottleneck.

Shorter or more clever code does not always mean faster compiler output. For example, folding pointer increments into the middle of complicated pointer arithmetic expressions makes nothing actually faster, but makes errors likely when the next programmer has to change the expression. If the target machine architecture has special automatic incrementing instructions, the compiler will emit them when appropriate even if you don’t use the increment operator explicitly.

For another example, `n >> 1` is no faster than `n / 2`. Regardless of machine architecture, any reasonable compiler will emit the fastest sequence of instructions available for either case. The choice of using one or the other should be based on whether an logical or arithmetic operation is meant to the human reading the program, or whether the program might one day be changed to `n >> 2` or `n / 3`.

3.1 – Use NULL

Use `NULL`, not `0`, to represent the null pointer. `NULL` must be defined as `0` or `(void *) 0`, but not `0L` or `0UL`, regardless of whether or not the value used by the compiler for the `NULL` pointer value has some bits set. The lexeme `0` is treated specially by the compiler when it occurs in the context of a pointer value. It is therefore completely legal, standard C to write `0` instead of `NULL`. The designer of the C++ language encourages writing null pointers as `0`, as he regards any use of the C preprocessor as infinitely costly. However, it looks strange to most programmers, as `0` reads as a number and not a pointer value.

3.2 – Take Compiler Warnings Seriously

Use all of the compiler’s advice. There is seldom any reason to leave a compiler warning unheeded. When using `gcc`, always compile `-Wall`. When not using `gcc`, incorporate a separate compile under `gcc -Wall` as part of the testing.

In whatever build system you use, do not process the output of compiler runs to suppress or even collate its messages. Programmers need the unvarnished truth.

3.3 – Omit Junk Compiler Hints

Directives such as `inline` and `register` are largely meaningless. Any reasonable optimizer will use registers for the most frequently used variables, and will inline short functions. However, if a function definition is put in a header file, declare it `static inline` to make its linkage clear.

If so much speed is required that the compiler must be given explicit instructions about how to optimize, only compiler specific features will control the optimizer in ways that amount to anything. The more generic features of `register` and `inline` will not, so avoid them in the first place.

3.4 – Avoid `#pragma`

The use of `#pragma` basically means you are no longer writing standard C99; you tie your code to a specific compiler or compilers. There is only one reason to use `#pragma`, and it is similar to the reason why you would use other special features of your compiler, such as explicit assembler sections. There must be a compelling need for some effect in the compiler output and no other way to achieve it before you should consider using `#pragma`.

Do not use `#pragma` once. Decent compilers will notice if a header is entirely wrapped in a `#ifndef` without the `#pragma`.

3.5 – Avoid Unspecific Integer Types

Use `stdint.h`. Avoid using `short`, and `long`. Instead, use types with specific sizes such as `int16_t`, `int32_t`, and `ptrdiff_t`.

Avoid `stdbool.h`. Use an `int` to store a boolean value, or at most replace `stdbool.h` with the single type declaration `typedef _Bool bool;`. The lexemes `bool`, `true`, and `false` intuitively work like keywords, but instead are made preprocessor macros by `stdbool.h`. This is a rude hack and the source of subtle bugs.

3.6 – Give Integer Literals Explicit Type

Write integer literals of type `int` as unadorned base ten numbers. Write all other literals either as hexadecimal numerals with the appropriate number of places, or as an explicit cast of an unadorned base ten num-

ber. Mark unsigned literals with final U, and pad with leading zeros to account for the full bit width. Avoid marking numeric literals with L or LL. Consider the following good example.

```
int a = 5;
uint32_t b = 0x00112233U;
int16_t c = (int16_t) -7;
```

3.7 – Do Not Perform Bitwise Operations On Signed Data

Compilers are free to generate whatever results they want as the result of the following expression.

```
signed int a = -7 >> 1;
```

In general, bitwise operations, `>>`, `<<`, `&`, `|`, `~`, and their analogous assignment operators, should happen only to unsigned integer types. To do otherwise is to tie your code to a specific compiler and execution platform. At worst, hide all your `HAKMEM` functions in a portability module.

3.8 – Do Not Assume `char` Types Are `unsigned`

`char` is mostly indistinguishable from other integer types. However, although the declaration `int i` declares a signed integer, `char a` has compiler-specific signedness. Most sensible compilers will make unadorned `char` the same as `unsigned char`, but do not rely on that. Often `uint8_t` is more appropriate than `char` anyway.

3.9 – Write C99 Code

At the time of this handbook’s publication, there are three important incarnations of the C language. These are the language described by Kernighan and Ritchie in their book “The C Programming Language” (1978, 1988 editions), which is designated “K&R C”, the first ANSI standardization of C, which is designated “C90”, and the second standardization, designated “C99”. Write C according to the most recent standardization.

3.9.1 – Do Not Use Archaic C Idioms

Significant changes have gone into C standards over the years. For a time in the early history of the C language it was important to write code that would compile on older standardizations of the language, as

newer standards-compliant compilers weren't available for all platforms. Many years have passed since then. There's no longer any excuse.

The most egregious example of an archaic idiom to be not only avoided, but excised immediately and at all costs, is the use of the C preprocessor to work around bad linkers. Early C linkers could only disambiguate names up to the first six characters, and in such environments it was commonplace to use C preprocessor macros to make longer names in code that compiled down into names that the linker could digest.

```
/* names matching 'stack_.*' all look the same
   to the linker */
#define stack_pop stkpop
#define stack_push stkpsh
#define stack_is_overflowed stkisv

extern node_t stack_pop();
extern node_t stack_push();
extern int stack_is_overflowed();
```

This seems like a good solution, as all of the compressed names are off in their own file where they can be otherwise ignored. But it's a complete nightmare when loaded into to the debugger.

3.9.2 – Do Not Write C++

There are lots of ways to make C code also compile as C++ code. For example, not using the identifier `new`, which is a keyword in C++, or disciplining variable initialization in various ways because the constructor may be expensive. It takes a lot of effort to write good C code that is also C++ code. C++ has extremely robust ways of linking to C code, so it isn't needed. Focus on writing good C code. Don't write C++ code.

The designer of the C++ programming language envisioned C++ as a “better C” – a language that C programmers could start using by continuing to develop mostly in C, but adding new features from C++ over time. Do not do this. It is impossible to add, for example, just namespaces, without also needing to comb through all of your existing code for C++ incompatibilities first. The result is both awful C code and awful C++ code.

3.9.3 – Do Not Target C

Avoid generating C code. Generated C code is difficult to read, and impossible to manage in the debugger. Good ways to generate C code integrate well into the debugger, such as the C preprocessor, `flex`, and `bison`. Bad ways to generate C code include `m4` and `perl`.

3.9.4 – Do Not Make Lint Annotations

Lint programs check C source for probable errors. Given that Lint will complain about some perfectly legitimate programs, they generally include robust means to suppress specific checks at specific points in source code, by annotating that source code with structured data in comments. Use Lint programs to audit source code, but do not annotate source code to explicitly acknowledge violations of Lint rules. For example, consider the following.

```
#include <stdio.h>
int foo(int a)
{
    printf("hello\n");
}
main()
{
    foo(3);
}
```

Both `gcc -Wall` and `splint` correctly find the problem where `foo` is defined to have a return value, but no return value is specified in the body of `foo` nor in its call from `main`. These tools also correctly find the unused argument to `foo`. However, `splint` suggests that the above code either be fixed or rewritten as follows, to annotate the unused variables

as intentionally so.

```
#include <stdio.h>
int foo(/*@unused@*/ int a)
{
    printf("hello\n");
}
main()
{
    (void) foo(3);
}
```

The idea is that by marking `a` as unused in the argument list of the definition, you've acknowledged this specific instance of the problem. Likewise, casting the return value of `foo` to `void` is a way of acknowledging `foo` has a return value that is purposefully ignored in this call. The uncast call to `foo` could mean carelessness.

There are two problems with this practice. First, programmers might be tempted to insert the Lint annotations for legitimate problems, rather than fixing them. In the above example, the actual problem is that `foo` should be declared as returning `void`, but the cast of the call to `foo` to `void` is sufficient to make Lint quiet down. The next programmer must remove or correct Lint annotations as the code changes, with little benefit.

Second, rewriting the code in this way satisfies `spint`, but not all such tools. As with any structured data in comments, it is a Faustian bargain. Other Lint programs may want the annotation to occur just before the function definition as `/* ARGSNOTUSED */`, for example. In general, it is always possible to fix a problem reported by a Lint program without using explicit suppression commands. Do so, or live with the lint output as is.

4 – RFC 1122

Libraries of code with documented entry points and data structures – so-called “Application Programmer Interfaces” – are much like Internet protocols. RFC 1122 gives excellent advice regarding how to devise and utilize Internet protocols. The most relevant advice in terms of library interfaces is the maxim “be conservative in what you send, and liberal in what you receive.”

4.1 – Do Not Pollute Namespaces

Use `.h` files to draw a minimal line around parts of a module or compilation unit that need to be exposed to other parts of the program. Wrap all `.h` file contents in a conditional compilation unit that will not expand if that `.h` file has been included previously, as in the following example.

```
#ifndef FOO_H_INCLUDED
#define FOO_H_INCLUDED

// ... contents of .h file here ...

#endif
```

The name used in this wrapper is another case where a name that no other module is using must be chosen, which is impossible to do perfectly. Choose carefully.

Including the header for one module should not require that another header be included first; instead, the header for a module should itself include everything it needs.

Avoid including headers that are not needed. It is difficult for the compiler to detect such problems. Include all headers from which names are used. For example, if the file `foo.h` declares the global variable `foo`, and the file `bar.h` includes the file `foo.h` and declares the global variable `bar`, include both `foo.h` and `bar.h` in files which use both variables `foo` and `bar`. Do not just include `bar`. In a file which just used the variable `bar`, it would be fine to just include `bar.h`, even though the overall output of the preprocessor would be similar. This minimizes side effects when modules change.

Include header files in order from most general to most specific. That is, include system and standard library headers first, then commonly used third-party library headers (such as a graphics toolkit), then headers from modules in code shared with other projects, then headers from

modules specific to the project itself. Likewise, write header files under the assumption that this is the order in which they will be included.

Consider names that begin with one or more underscores to be the sole domain of the compiler and standard library. Do not pollute this namespace.

4.2 – Declare Function Arguments

Declare or define all functions before they are used. Declare all arguments with types and names, and declare functions that have no arguments using the `void` argument list, as in the following good examples.

```
extern int foo(int in, int *out);
extern uint16_t thunk(void);
```

The way in which function arguments are presented in function declarations has changed significantly over the lifetime of the C language, and has also diverged significantly from the semantics for function declarations in C++, making the issues confusing. In essence, the C standard permits three things which shouldn't be done. The first is to omit the argument list entirely from the declaration, as in the following bad versions of the above good examples.

```
extern int foo();
extern uint16_t thunk();
```

There is a good argument for doing things this way, as it eliminates what amounts to duplicating code from the definition into the declaration. However, in the case of external functions declarations, such as are in header files, such code can not really be considered duplication. Also, there is potential confusion between the syntactically empty argument list `()`, which means the argument list is arbitrary, and the semantically empty argument list `(void)`, which means no arguments are permitted. Decent compilers can catch problems with argument type mismatch, but only when argument types are declared.

The second is to include type names in the argument list, but to leave the argument names out, as in the following bad versions of the above good examples.

```
extern int foo(int, int *);
extern uint16_t thunk(void);
```

This has the pleasing effect of making `(void)` argument lists, which can not have a name, more orthogonal in appearance to the others.

But it also eliminates potentially useful information from the declaration. Names of arguments may be all others ever bother to read about your function. If argument names (not types) are mismatched between declarations and definitions, most compilers won't care, making code duplication issues no worse for including the names.

The third is to simply fail to declare functions before they are used. As described above, empty argument lists just indicate arbitrary arguments are possible, and undeclared function return types default to `int` which is often wide enough to hold a pointer, thereby making calls to the function basically work for certain return types. Do not do this. Bugs introduced by such lack of declaration can often be too subtle to notice.

4.3 – Map Inputs, Outputs, And Exceptions Properly Onto Arguments And Returns

`malloc` either returns the desired results, or `NULL`. Use `malloc` style exception reporting only when there is only ever one possible exception to report, and when there is a return value such as `NULL` that can not possibly come up as a legitimate result.

Some system library routines either return the desired results, or a special value indicating that some exception occurred. On return from such a routine, the global variable `errno` will be set to a code number that indicates what happened. System libraries go through a lot of work to make `errno` work correctly among multiple threads. It may or may not be possible to write programs that do similar things as those system libraries. Do not use `errno` style error reporting. Instead, pass both inputs and pointers to storage for outputs in as parameters, and use the return value solely for signaling error conditions.

Order function arguments as logically and unsurprisingly as possible. If a function's arguments include pointers to storage for results, put those arguments at the end of the list. Readers generally assume inputs will appear to the left, and outputs to the right, in an argument list.

4.4 – Provide Interfaces To Complex Data Structures

There are three means to pack data structures in C that can become complicated. These are bitfields, casting structure pointers to pointers to the type of their first component, and structures ending in variable length arrays. These kinds of complex packing schemes can make it so small changes in a data structure can trigger a cascade of unavoidable code changes. Write interfaces to these complex data structures using

the C Preprocessor. Using the preprocessor, and not C functions, allows abstractions that can appear on the left hand side of an assignment. Consider the following good example.

```
typedef struct packet_s {
    uint16_t type;
    uint16_t length;
    unsigned char data[];
} packet;

#define TYPE_OF(P) ((P).type)
#define LENGTH_OF(P) ((P).length)
#define DATA_OF(P, N) ((P).data[(N)])
```

Notice that all three macros are legal on the left hand side of an assignment, as in `TYPE_OF(P) = 5`. Now imagine that this structure needs another variable length field added later, in between the type/length header and the packet data. All that needs to change is the interface definitions, as in the following example.

```
typedef struct packet_s {
    uint16_t type;
    uint16_t length;
    uint16_t route_length;
    // data holds first route then payload
    unsigned char data[];
} packet;

#define TYPE_OF(P) ((P).type)
#define LENGTH_OF(P) ((P).length)
#define DATA_OF(P, N) ((P).data[(N) + (P).route_length])
#define ROUTE_LENGTH_OF(P) ((P).route_length)
#define ROUTE_DATA_OF(P, N) ((P).data[(N)])
```

If a complex data structure has natural operations other than just setting and getting its constituents, consider also providing even fuller interfaces to such complex data structures, using C functions instead of macros.

4.5 – Write Reentrant Functions

Write functions which are reentrant. Avoid `static` variables in functions, global variables, and functions with side effects in general. Do

not write functions with surprising side effects. Expect that all functions of return type `void` have side effects. Likewise, write functions with important side effects as returning `void`.

4.6 – Do Not Hack In Portability

In the following bad example, portability has been hacked in.

```
#ifdef SYSTEM_A
#define CORGE_TEST() (corge() == 3)
#else
#if defined(SYSTEM_B) || defined(SYSTEM_C)
#define CORGE_TEST() (corge() == 7)
#else
#error Unknown system.
#endif
#endif
    if (CORGE_TEST()) {
        gralt();
    }
```

Portability hacks like these can explode in complexity as the porting targets, and the subtle relationships between them, increase over time. For all but the most trivial portability hacks, use separate modules of code. Make the decision of which port to target in the build system by selecting the appropriate modules for that port.

4.7 – Assume There Is A Debugger

Assume there is a symbolic debugger with breakpoints and the ability to examine variable values. Do not litter the code with debugging log

commands, as in the following bad example.

```
#ifndef debug
#define debug(...) fprintf(stderr, __VA_ARGS__)
#endif
int frob_bits(int a)
{
    int i = 0;
    debug("a = %d\n", a);
    for (assert(i == 0); i < 30; i++) {
        debug("got here\n");
        a = quux_bits(i, a);
    }
    return (a);
}
```

The messages above would be meaningless to all but the programmer who just wrote them. It is far better to run the program under a proper debugger with a breakpoint on `frob_bits`, or if the messages were of general use even outside of the context of debugging, to use a proper logging facility.

There may be rare cases on embedded systems in which there is no debugger, but then it is equally unlikely that there will be a back channel for a debug log.

5 – Do Not Fear The Preprocessor

The C preprocessor is easily misused. Most C programming advice includes dire warnings about how horrible the preprocessor is, and how it should be avoided at all costs. Use the preprocessor in careful and deliberate ways to add clarity and remove duplicated code. Avoid using the preprocessor when it hides data from the debugger.

5.1 – Use Functions To Factor

Prefer functions to macros for factoring significant sections of repeated code into a single syntactic entity. Give such helper functions static linkage and then trust the compiler to optimize away any associated function call overhead. Take the following code for example, in which the sections marked X are both the same. We would like all the code in the two identical sections X to occur only once in the source code, but without changing the type and number of arguments to `foo` and `bar`.

```
int foo(int a)
{
    int c = 3;
    // X
}
int bar(int a)
{
    int c = 17;
    // X
}
```

There are essentially three ways to eliminate the repeated section X. The worst is to make `c` a global variable. The next worse is to use a

macro as follows.

```
#define DEFGRALT(F, C) \
int (F)(int a) \
{ \
    int c = (C); \
    // X \
}
DEFGRALT(foo, 3);
DEFGRALT(bar, 17);
```

The last is to use helper functions, which is most preferable of all.

```
static int help_foo_bar(int a, int c)
{
    // X
}
int foo(int a)
{
    help_foo_bar(a, 3);
}
int bar(int a)
{
    help_foo_bar(a, 17);
}
```

5.2 – Prefer enum Over #define

Use `enum` to name collections of integer constants with all different values, especially those that appear as cases in `switch case` statements. Consider the following good example.

```
typedef enum place_type_e {
    farm, house, factory
} place;
```

If `farm`, `house`, and `factory` were preprocessor definitions, the compiler would not be able to, for example, warn of case statements that fail to handle the `factory` case.

Use the preprocessor to name arbitrary numeric constants, especially floating point constants and integers that are used to size arrays. Con-

sider the following bad examples.

```
const float ratio_magic = 4.789736;
enum constants {
    max_quads = 6,
    max_shoes = 6
};
```

If `ratio_magic` were a preprocessor define, the name would never make it to the debugger. As a variable in the debugger, you could even set a watch point on it. However, the optimizer can not inline `ratio_magic` as if it were a literal; there must be a single storage location for the value of a variable like this, even when declared `const`.

It is possible to use constants named in an `enum` to size arrays and otherwise name integer literals, but each enum value with the same integer value is indistinguishable in the debugger. For the code above, the debugger is as likely to report a value of `max_quads` as it is a value of `max_shoes` for any variable of type `enum constants`, since they both mean 6.

5.3 – Write Macros That Work Like Expressions Or Statements

Write macros that behave syntactically like an expression or a statement. Do not write macros that extend the syntax of the language. Following is a bad example of such an extension.

```
{
    TRY
        int i = 0;
        for (assert(i == 0), i < 5, i++) corge(i);
        galt();
    CATCH(foo)
        bar();
    ENDTRY
}
```

Above, `TRY` and `CATCH` are meant to add C++ style exception handling to the language, perhaps by making calls to the system library's `setjmp` and `longjmp` routines. These macros are very brittle. Depending on the implementation, the `int i` declaration may cause the `TRY` macro to fail, as might zero or more than one `CATCH` sections, or a failure to put `ENDTRY` at the end of its block. The `THROW` macro we can assume is used in the `corge` and `galt` functions probably has side effects which are not thread safe.

Following is another bad example.

```

pants p = pants_new(5);
FOR_EACH_PANTS(p) {
    pants_guess(p);
    pants_print(p);
}

```

Presumably the macro `FOR_EACH_PANTS` expands to the head of a `for` statement. If it doesn't, the reader had better know about it, as it might fail with a single statement instead of a block, and so forth. The abstraction can not be made strong enough to both work correctly and hide enough from the eventual user. Instead, avoid the abstraction entirely.

Another bad example of using the preprocessor to change the language syntax follows.

```

#define FUNC_PTR(RT, FP) RT (* FP)
// declare foo to be pointer to function
// returning int, taking int
FUNC_PTR(int, foo)(int a);

```

Declaring pointers to functions has syntax which requires the use of parenthesis to associate the pointer type indicator with the name of the function, and not the return type of the function. Some programmers find this so objectionable that they hide the syntax behind a macro. The above `FUNC_PTR`, aside from being a terrible name, can not be used to declare an array of pointers to functions, for example.

Assume that macros which work like expressions might produce their arguments more than once, or not at all, when expanded by the C preprocessor. Use explicit temporary variables around the use of such a macro to control evaluation. Conversely, assume that macros which behave like statements will produce each of their arguments exactly once. Write macros to match these expectations.

To make statement-like macros produce their arguments only once, it is often necessary to introduce a scope for temporary variables. Do this by wrapping the body of the macro in `do while (0)`, as in the

following example.

```
#define CORGE(X) \
do { \
    int x = (X); \
    int i = f(x); \
    h(i + x, g(i)); \
} while (0)
```

This will allow the macro to be used in the same syntactic positions as any other statement, including as the single statement in the body of an `if else` statement, without causing any problems.

5.4 – Wrap Produced Expressions In Extra Parenthesis

Wrap arguments to macros with extra parenthesis in the macro body. This prevents errors where macros produce syntactically legal but semantically incorrect code for expression arguments. Also, when writing a macro that acts as an expression, wrap the entire result in parenthesis for similar reasons. Here are bad examples.

```
#define TIMES_TWO(N) N * 2
#define PLUS_ONE(N) N + 1
int a = 1;
int b = TIMES_TWO(a + 3);
int c = PLUS_ONE(a) * 2;
```

Presumably, the intent was for `b` to be initialized to eight, but instead it will be initialized to seven. Also, `c` will be initialized to three, not four. In both of these cases, the intuition of the reader is that the macro evaluates its arguments like a function call. Adapt your macros to that intuition. The following macros use additional parenthesis to cause their arguments to be evaluated with the correct precedence, making them work more like function calls.

```
#define TIMES_TWO(N) ((N) * 2)
#define PLUS_ONE(N) ((N) + 1)
```

5.5 – Put Conditional Compilation Breaks Between Statements

Avoid putting preprocessor conditional directives in the middle of statements. It is okay to use an extra temporary variable, an extra `#define` or to repeat half a line of code, to avoid strange conditional compilation forms, as in the following bad example.

```
extern void foo(int a);
extern void bar(int a);
// ...
#ifdef FOO
    foo
#else
    bar
#endif
    (3);
```

Separating the function from its arguments among conditional compilation directives makes for extremely difficult reading. A better way to write this would be as follows.

```
#ifdef FOO
    foo(3);
#else
    bar(3);
#endif
```

This is still brittle, since changing 3 to 4 must be done in (at least) two places. The best way to write this is as follows.

```
#ifdef FOO
#define CORGE(X) foo(X)
#else
#define CORGE(X) bar(X)
#endif
    CORGE(3);
```

5.6 – Do Not Abuse ## and \

Assume that the debugger will not see C preprocessor names, but rather only what the C preprocessor expands into. Constructing names with ## and/or using #define instead of enum makes the debugger input and the source code difficult to reconcile.

Break lines in definitions only between tokens, preferably only between statements and/or expressions. Never end a line in the middle of a token using \.

5.7 – Use X-Macros To Avoid Duplicating Code

When data is duplicated in several locations of code such that all location must be updated when a change is needed, write the data once into a table which can be used by the C preprocessor to generate the needed code. Name the file with the suffix `.def`.

```
#ifndef X
#error X not defined for X-Macro file.
#else
X(salad, 8.95)
X(pig, 3.35)
X(peas, 0.99)
X(pasta, 2.50)
X(paste, 0.50)
#endif
```

To use the file, `#define X` to be a macro that generates the appropriate code based on its arguments. Then `#include` the `.def` file. Finally, `#undef X`.

```
#define X(A, B) a,
typedef enum item_e {
#include "menu.def"
} item_kind;
typedef struct menu_s {
    item_kind item;
    char *name;
    float price;
} menu;
#include "menu.def"
#define X(A, B) {a, #A, b},
menu items[] = {
#include "menu.def"
}
#undef X
```


6 – Choose Tools Wisely

Use these recommended tools. Using other than these tools means putting up with those who do, without complaining.

6.1 – GNU Toolchain

GCC, GDB, and GNU Binutils are often collectively referred to as the “GNU Toolchain.” Use the GNU Toolchain unless it does not target the required platforms, or unless there is a measurable and significant performance gain in using another tool. Use GCC and GNU Binutils for baseline testing, even when compiled results are to be made with different tools, as the error reporting in from these programs is well vetted and well understood.

6.2 – Valgrind

Valgrind is an ideal compliment to the GNU Toolchain, providing indispensable tools for use during testing, debugging, and profiling.

6.3 – Emacs

Prefer to use Emacs front-ends to the remainder of these tools, such as `vc`, `ediff`, `compile`, `cscope`, and `gud`. Use Emacs subsystems to browse and edit code, such as `ecb`, `etags`, `completion`, and `yasnippet`.

6.4 – Git

Distributed version control is essential. Prefer to make lots of small, well annotated commits rather than large changes. Keep separate lines of development in separate branches.

6.5 – GNU Make

There are so many edge cases in which GNU Make wins over other versions of `make`, that it is usually easier to port GNU Make than to write “portable” Makefiles. GNU Make specific features are therefore also recommended.

Write one `Makefile`. Do not write a second `Makefile`, or run `make` recursively. Instead, stop everything and read Peter Miller’s “Recursive Make Considered Harmful” (<http://cj5.info/pmiller/books/rmch>) again.

6.6 – C A Reference Manual

The best, most thorough, and most useful book on the C language is “C A Reference Manual” by Harbison and Steele. A copy of the latest edition is indispensable.

6.7 – The CERT C Secure Coding Standard

The “CERT C Secure Coding Standard” (CCSCS) describes itself as follows.

The CERT C Secure Coding Standard provides rules and recommendations for secure coding in the C programming language. The goal of these rules and recommendations is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. The application of the secure coding standard will lead to higher-quality systems that are robust and more resistant to attack.

The CCSCS reads like a laundry list of problems which have bitten experienced C programmers over the years, each elegantly described, factored, and numbered into a well composed standard. Although the point of the standard is better security, the rules and recommendations it gives are generally useful for writing robust C programs.

6.8 – L^AT_EX

Although `texinfo` produces documentation for which there is a good browser in Emacs, L^AT_EX is far more robust. Also, it is generally more productive to violently chop at L^AT_EX than it is to gently grow plain T_EX. Write documentation in L^AT_EX and publish it in PDF format.

6.9 – Tools To Avoid

Avoid portability hacks like `Imake`, `autoconf/automake`, and `libtool`. The UNIX workstation craze is long since over, and we only have a handful of possible configurations to manage. We can go back to using `make` and `configuration.h` now. Do not waste any more time debugging `m4` macros.

Avoid code indexing tools other than `etags`, such as `id-tools` and `Global`. These don’t offer any features beyond `etags` to Emacs users. All such tools are going to give you untrustworthy results anyway, unless someday someone writes one that does non-trivial analysis of C Preprocessor macros.

Epilogue

Here is the code from the start of this Handbook, rewritten to conform to all given specific rules.

```
main(int argc, char **argv)
{
    int i = 0;

    argc--;
    if (argc) {
        main(argc, argv);
        // sign bit of 0th character can be used
        // without otherwise wrecking it
        argv[argc][0] *= -1;
        for (i = 1; argv[i] != 0; i++) {
            if (argv[i][0] > 0) puts(argv[i]);
        }
        puts("-- ");
        main(argc, argv);
    }
}
```

Although this is significant progress, the meaning of the code is still not clear. No amount of rules can force a programmer to write good programs. Hopefully, this handbook will at least point in the right direction, and underscore the important issues.

