C and C++ Elements to Avoid

Shlomi Fish <shlomif@cpan.org>

Table of Contents

Introduction

Often when people ask for help with C or C++ code, they show code that suffers from many bad or outdated elements. This is expected, as there are many bad tutorials out there, and lots of bad code that people have learned from, but it is still not desirable. To encourage best practices, here is a document of some of the common bad elements that people tend to use and some better practices that should be used instead.

A book I read said, that as opposed to most previous idea systems, they were trying to **liquidate negatives** instead of to instil positives in people. So in the spirit of liquidating negatives, this tutorial-in-reverse aims to show you what **not to do**.

Note: Please don't think this advice is meant as gospel. There are some instances where one can expect to deviate from it, and a lot of it can be considered only the opinion of its originators. I tried to filter the various pieces of advice I found in the sources and get rid of things that are either a matter of taste, or not so critical, or that have arguments for and against (so-called colour of the bike shed arguments [http://bikeshed.com/]), but some of the advice here may still be controversial.

The List of Bad Elements

No Indentation

Indentation [http://en.wikipedia.org/wiki/Indent_style] means that the contents of every block are promoted from their containing environment by using a shift of some space. This makes the code easier to read and follow.

Code without indentation is harder to read and so should be avoided. The Wikipedia article [http://en.wikipedia.org/wiki/Indent_style] lists several styles - pick one and follow it.

Compiling without warnings flags

C and C++ Compilers have flags to toggle on warnings such as `-Wall`, `-Wextra`, or `-Weverything`. It is a good idea to specify as many of them as possible when compiling the code and to fix the warnings where appropriate.

Someone gave me this GCC warnings’ theme:

```
#!/bin/bash
gcc \
    -std=c99 \
    -ansi \
    -pedantic \
    -W \
    -Wall \
```

```
-Wbad-function-cast \
-Wcast-align \
-Wcast-qual \
-Wdeclaration-after-statement \
-Wfloat-equal \
-Wformat-nonliteral \
-Winline \
-Wmissing-declarations \
-Wmissing-prototypes \
-Wnested-externs \
-Wold-style-definition \
-Wpointer-arith \
-Wshadow \
-Wstrict-prototypes \
-Wundef \
-Wunused \
-Wwrite-strings
```

Calling variables "file"

Some people call their variables "file". However, file can mean either file handles [http://en.wikipedia.org/wiki/File_descriptor], file names, or the contents of the file. As a result, this should be avoided and one can use the abbreviations "fh" for file handle, or "fn" for filenames instead.

Identifiers without underscores

Some people name their identifiers as several words all in lowercase and not separated by underscores ("_"). As a result, this makes the code harder to read. So instead of:

```
char * namesofpresidents[NUM_PRESIDENTS];
```

Say:

```
char * names_of_presidents[NUM_PRESIDENTS];
```

Or maybe:

```
char * presidents_names[NUM_PRESIDENTS];
```

Write code in Paragraphs using Empty Lines

If one of your blocks is long, split it into "code paragraphs", with empty lines between them and with each paragraph doing one thing. Then, it may be a good idea to precede each paragraph with a comment explaining what it does, or to extract it into its own function or method.

Don't start Classes with a Lowercase Letter

In C++, classes should start with an uppercase letter (see the Wikipedia article about letter case [http://en.wikipedia.org/wiki/Letter_case]) and starting them with a lowercase letter is not recommended.

```
# Bad code
```

```
class my_class
{
    .
    .
    .
};


class MyClass
{
    .
    .
    .
};
```

Avoid Intrusive Commenting

Some commenting is too intrusive and interrupts the flow of reading the code. Examples for that are the `//////////////` or `/******************/` hard-rules that some people put in their code, the comments using multiple slashes like `///`, or excessively long comment block. Please avoid all those.

Some schools of software engineering argue that if the code's author feels that a comment is needed, it usually indicates that the code is not clear and should be factored better (like extracting a method or a subroutine with a meaningful name.). It probably does not mean that you should avoid writing comments altogether, but excessive commenting could prove as a red flag.

If you're interested in documenting the public interface of your modules and command-line programs, refer to tools such as Doxygen [http://en.wikipedia.org/wiki/Doxygen], which may prove of use.

Accessing Object Slots Directly

It is a bad idea to access the slots/properties/members of an object or a pointer to it directly. E.g:

```
# Bad code


obj->my_slot = 5;
.
.
.
if (my_obj.my_boolean_slot)
{
}
```

Instead, create accessors, readers and writers - see mutator method [http://en.wikipedia.org/wiki/Mutator_method] on the Wikipedia and the class Accessors [http://perl-begin.org/tutorials/perl-for-newbies/part5/#page--accessors--DIR] section of the Perl for Newbies tutorial.

'^' and '$' in Regular Expressions

Some people use "^" and "$" in regular expressions to mean beginning-of-the-string or end-of-the-string. However, they can mean beginning-of-a-line and end-of-a-line respectively using the `/m` flag which is

confusing. It's a good idea to use `\A` for start-of-string and `\z` for end-of-string always (assuming they are supported by the regex syntax), and to specify the `/m` flag if one needs to use "^" and "$" for start/end of a line.

Magic Numbers

Your code should not include unnamed numerical constants also known as "magic numbers" or "magic constants" [http://en.wikipedia.org/wiki/Magic_number_%28programming%29#Unnamed_numerical_constants]. For example, there is one in this code to shuffle a deck of cards:

```
# Bad code


for (int i = 0; i < 52; i++)
{
    const int j = i + rand() % (52-i);
    swap(cards[i], cards[j]);
}
```

This code is bad because the meaning of 52 is not explained and it is arbitrary. A better code would be:

```
const int deck_size = 52;

for (int i = 0; i < deck_size; i++)
{
    int j = i + rand() % (deck_size - i);
    swap(cards[i], cards[j]);
}
```

Mixing Tabs and Spaces

Some improperly configured text editors may be used to write code that, while indented well at a certain tab size looks terrible on other tab sizes, due to a mixture of tabs and spaces. So either use tabs for indentation or make sure your tab key expands to a constant number of spaces. You may also wish to make use of auto-formatters like GNU indent [http://en.wikipedia.org/wiki/Indent_%28Unix%29] to properly format your code.

Several synchronised arrays.

Related to “varvarname” is the desire of some beginners to use several different arrays with synchronised content, so the same index at every array will contain a different piece of data for the same record:

```
# Bad code


char * names[ITEMS_COUNT];
char * addresses[ITEMS_COUNT];
int ages[ITEMS_COUNT];
char * phone_numbers[ITEMS_COUNT];

.
.
```

```
.

names[num] = strdup("Isaac Newton");
addresses[num] = strdup("10 Downing St.");
ages[num] = 25;
phone_numbers[num] = strdup("123456789");
```

These arrays will become hard to synchronise, and this is error prone. A better idea would be to use an array (or a different data structure) of structs, classes, or pointers to them:

```
Person * people[ITEMS_COUNT];

num = 0;
people[num++] = create_person(
    "Isaac Newton",
    "10 Downing St.",
    25,
    "123456789"
);
```

Modifying data structures while iterating through them.

Some people ask about how to add or remove elements to an existing array or a different container when iterating over them using loops. The answer to that is that it likely won't be handled too well, and it expects that during loops the keys of a data structure will remain constant.

The best way to achieve something similar is to populate a new container during the loop. So do that instead.

Comments and Identifiers in a Foreign Language

Apparently, many non-native English speakers write code with comments and even identifiers in their native language. The problem with this is that programmers who do not speak that language will have a hard time understanding what is going on here, especially after the writers of the foreign language code post it in to an Internet forum in order to get help with it.

Consider what Eric Raymond wrote in his "How to Become a Hacker" document [http://www.catb.org/~esr/faqs/hacker-howto.html#skills4] (where hacker is a software enthusiast and not a computer intruder):

> 4. If you don't have functional English, learn it.
>
> As an American and native English-speaker myself, I have previously been reluctant to suggest this, lest it be taken as a sort of cultural imperialism. But several native speakers of other languages have urged me to point out that English is the working language of the hacker culture and the Internet, and that you will need to know it to function in the hacker community.
>
> Back around 1991 I learned that many hackers who have English as a second language use it in technical discussions even when they share a birth tongue; it was reported to me at the time that English has a richer technical vocabulary than any other language and is therefore simply a better tool for the job. For similar reasons, translations of technical books written in English are often unsatisfactory (when they get done at all).

> Linus Torvalds, a Finn, comments his code in English (it apparently never occurred to him to do otherwise). His fluency in English has been an important factor in his ability to recruit a worldwide community of developers for Linux. It's an example worth following.
>
> Being a native English-speaker does not guarantee that you have language skills good enough to function as a hacker. If your writing is semi-literate, ungrammatical, and riddled with misspellings, many hackers (including myself) will tend to ignore you. While sloppy writing does not invariably mean sloppy thinking, we've generally found the correlation to be strong — and we have no use for sloppy thinkers. If you can't yet write competently, learn to.

So if you're posting code for public scrutiny, make sure it is written with English identifiers and comments.

“using namespace std;”

One can often see C++ code with `using namespace std;` on top, but that is a bad idea. The C ++ standard requires that the “std” namespace contain certain symbols, but it doesn't prevent it from containing **other** symbols (presumably, implementation details of the standard library). If you use `using namespace std;`, you never know what else you might also be bringing into the global namespace.

One possible alternative is to selectively do `using std::cout;`, `using std::string;` and so forth, for each symbol that you wish to use.

The Law of Demeter

See the Wikipedia article about “The Law of Demeter” [http://en.wikipedia.org/wiki/Law_of_Demeter] for more information. Namely, doing many nested method calls like `obj->get_employee('sophie')->get_address()->get_street()` is not advisable, and should be avoided.

A better option would be to provide methods in the containing objects to access those methods of their contained objects. And an even better way would be to structure the code so that each object handles its own domain.

Passing parameters in delegation

Sometimes we encounter a case where subroutines each pass the same parameter to one another in delegation, just because the innermost subroutines in the call-stack need it.

To avoid it, create a class, and declare methods that operate on the fields of the class, where you can assign the delegated arguments.

Duplicate Code

As noted in Martin Fowler's "Refactoring" [http://www.shlomifish.org/philosophy/books-recommends/#refactoring] book (but held as a fact for a long time beforehand), duplicate code [http://en.wikipedia.org/wiki/Duplicate_code] is a code smell, and should be avoided. The solution is to extract duplicate functionality into subroutines, methods and classes.

Long Functions and Methods

Another common code smell is long subroutines and methods [http://c2.com/cgi/wiki?LongMethodSmell]. The solution to these is to extract several shorter methods out, with meaningful names.

Using the ternary operator for side-effects instead of if/else

People who wish to use the ternary inline- conditional operator (`? :`) for choosing to execute between two different statements with side-effects instead of using `if` and `else`. For example:

```
# Bad code


cond_var ? (hash["if_true"] += "Cond var is true")
          : (hash["if_false"] += "Cond var is false")
```

(This is assuming the ternary operator was indeed written correctly, which is not always the case).

However, the ternary operator is meant to be an expression that is a choice between two values and should not be used for its side-effects. To do the latter, just use `if` and `else`:

```
if (cond_var)
{
    hash["if_true"] += "Cond var is true";
}
else
{
    hash["if_false"] += "Cond var is false";
}
```

This is safer, and better conveys one's intentions.

For more information, refer to a relevant thread on the Perl beginners mailing list [http://www.nntp.perl.org/group/perl.beginners/2012/04/msg120480.html] (just make sure you read it in its entirety).

Using Global Variables as an Interface to the Module

While it is possible to a large extent, one should generally not use global variables as an interface to a module, and should prefer having a procedural or an object oriented interface instead.

Using Global Variables or Function-"static" Variables

It is a good idea to avoid global variables or static variables inside functions; at least those that are not constant. This is because using such variables interferes with multithreading [http://en.wikipedia.org/wiki/Thread_%28computing%29#Multithreading], re-entrancy and prohibits instantiation. If you need to use several common variables, then define an environment struct or class and pass a pointer to it to each of the functions.

Declaring all variables at the top (Pre-declarations)

If you are writing C++ or C starting from the C99 standard onwards (which isn't supported by some non-standard-compliant compilers such as Microsoft's Visual C++), then you should avoid declaring all the variables at the top of the program or the subroutine (a practice that is referred to as "pre-declaration") and instead declare them when they are needed and with an immediate initialisation/definition using the = operator.

E.g:

```
# Bad code


int main(int argc, char * argv[])
{
    const char * name;
    int i;

    name = "Rupert";
    for (i=1 ; i<=10 ; i++)
    {
        printf ("Hello %s - No. %d!\n", name, i);
    }

    return 0;
}
```

Should be replaced with:

```
int main(int argc, char * argv[])
{
    const char * const name = "Rupert";
    for (int i=1 ; i<=10 ; i++)
    {
        printf ("Hello %s - No. %d!\n", name, i);
    }

    return 0;
}
```

Trailing Whitespace

With many editors, it can be common to write new code or modify existing one, so that some lines will contain trailing whitespace, such as spaces (ASCII 32 or 0x20) or tabs characters. These trailing spaces normally do not cause much harm, but they are not needed, harm the code's consistency, may undermine analysis by patching/diffing and version control tools. Furthermore, they usually can be eliminated easily without harm.

Here is an example of having trailing whitespace demonstrated using the `--show-ends` flag of the GNU cat command [https://en.wikipedia.org/wiki/Cat_%28Unix%29]:

```
> cat --show-ends toss-coins.pl
#!/usr/bin/perl$
$
use strict;$
use warnings;$
$
my @sides = (0,0);$
$
my ($seed, $num_coins) = @ARGV;$
$
```

```
srand($seed);  $
$
for my $idx (1 .. $num_coins)$
{$
    $sides[int(rand(2))]++;$
    $
    print "Coin No. $idx\n";$
}$
$
print "You flipped $sides[0] heads and $sides[1] tails.\n";$
>
```

While you should not feel bad about having trailing space, it is a good idea to sometimes search for them using a command such as `ack '[ \t]+$'` (in version 1.x it should be `ack -a '[ \t]+$'`, see ack [http://beyondgrep.com/]), and get rid of them.

Some editors also allow you to highlight trailing whitespace when present. See for example:

- Highlight unwanted spaces in Vim [http://vim.wikia.com/wiki/Highlight_unwanted_spaces]. Also see this post [http://vim.wikia.com/wiki/Highlight_unwanted_spaces].
- EmacsWiki: Show White Space [http://emacswiki.org/emacs/ShowWhiteSpace].

Finally, one can check and report trailing whitespace using the following CPAN modules:

- Test::EOL [http://metacpan.org/module/Test::EOL].
- Test::TrailingSpace [http://metacpan.org/module/Test::TrailingSpace].

Code and Markup Injection

Care must be taken when constructing statements that are passed to an interpreter, when putting arbitrary strings inside (using substring expansion or other methods). This is because if the strings are subject to input from the outside world (including the users), then one can use specially crafted strings for executing arbitrary commands and exploiting the system.

An example of this is outputting HTML using `fprintf(file_handle, "<p>%s</p>\n", paragraph_text);` or `my_file_fh << "<p>" << paragraph_text << "</p>" << std::endl;` which may allow inserting arbitrary, malicious, markup inside `paragraph_text`, which may include malicious JavaScript, that can steal passwords or alter the page's contents.

For more information, see:

1. "Code/Markup Injection and Its Prevention" [http://perl-begin.org/topics/security/code-markup-injection/] resource on the Perl beginners site.
2. Wikipedia articles about SQL injection [http://en.wikipedia.org/wiki/SQL_injection] and Cross-site scripting [http://en.wikipedia.org/wiki/Cross-site_scripting].
3. The site Bobby Tables [http://bobby-tables.com/] about SQL injections.

Using Undeclared Symbols

Some C or C++ compilers allow one to use functions or variables that have not been declared, while automatically inferring their types to be "int" or whatever. However, for good measure, all variables and functions should be declared with a proper type because they are often not the same as the compiler's

guess. One can configure GCC and similar compilers to emit an error on such cases using the `-Werror=implicit-function-declaration` flag, which is recommended to add to one's build system.

Declarations not in common headers

When declaring external functions, variables, classes, etc. make sure to put them in a common header file, which will also be included by the .c or .cpp file actually defining the common resource. This way if their type is changed, you will get a compiler error in the module that defines it, and there will be more certainty that there isn't a type mismatch.

Headers without #include guards or #pragma once

You should add #include guards [http://en.wikipedia.org/wiki/Include_guard], or the less standard but widely supported #pragma once [http://en.wikipedia.org/wiki/Pragma_once] into header files ("*.h" or "*.hpp" or whatever) to prevent them from being included times and again by other "#include" directives. Otherwise, it may result in compiler warnings or errors.

Overly Long Lines in the Source Code

It is a good idea to avoid overly long lines in the source code, because they need to be scrolled to read, and may not fit within the margins of your co-developers' text editors. If the lines are too long, you should break them or reformat them (for example, by adding a newline before or after an operator), and by breaking long string constants into several lines using the string concatenation operator `- ..`

Many coding standards require lines to fit within 80 characters or 78 characters or so, and you should standardise on a similar limit for your own code.

Regular Expressions starting or ending with ".*"

It is not necessary to put `.*` or `.*?` into the beginning or end of regular expressions to match something anywhere inside the string. So for example `regcomp(&regex, ".*ab+c")` can be replaced with the simpler: `regcomp(&regex, "ab+c")`. If you wish to match and extract the prefix, you should say `(.*?)` or `(.*)`.

Populating a Data Structure with Multiple Copies of the Same Pointer or Reference

You can sometimes see code like that:

```
# Bad code


int * my_array[NUM];

int * sub_array = malloc(sizeof(sub_array[0]) * SUB_NUM);
if (! sub_array)
{
    /* Handle out-of-memory */
}
for (int i = 0 ; i < NUM ; i++)
{
```

```
    populate_sub_array(i, sub_array);
    my_array[i] = sub_array;
}
```

The problem with code like this is that the same physical memory location is being used in all places in the array, and so they will always be synchronised to the same contents.

As a result, the code excerpts should be written as such instead:

```
int * my_array[NUM];

for (int i = 0 ; i < NUM ; i++)
{
    int * sub_array = malloc(sizeof(sub_array[0]) * SUB_NUM);
    if (! sub_array)
    {
        /* Handle out-of-memory */
    }
    populate_sub_array(i, sub_array);
    my_array[i] = sub_array;
}
my @array_of_arrays = map { [] } (1 .. $num_rows);
```

Using One Variable for Two (or More) Different Purposes

Within the scope of its declaration, a variable should serve one purpose, and serve it well. One should not re-use a variable for a completely different purpose later on in the scope.

Premature Optimisation

On various online forums, we are often getting asked questions like: “What is the speediest way to do task X?” or “Which of these pieces of code will run faster?”. The answer is that in this day and age of extremely fast computers, you should optimise for clarity and modularity first, and worry about speed when and if you find it becomes a problem. Professor Don Knuth had this to say about it:

> The improvement in speed from Example 2 to Example 2a is only about 12%, and many people would pronounce that insignificant. The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being practiced by penny-wise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs. In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal; and I believe the same viewpoint should prevail in software engineering. Of course I wouldn't bother making such optimizations on a one-shot job, but when it's a question of preparing quality programs, I don't want to restrict myself to tools that deny me such efficiencies.
>
> There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

(Knuth reportedly attributed the exact quote it to C.A.R. Hoare).

While you should be conscious of efficiency, and the performance sanity of your code and algorithms when you write programs, excessive and premature micro-optimisations are probably not going to yield a major performance difference.

If you do find that your program runs too slowly, refer to our resources about Optimising and Profiling code [http://c-begin.wikia.com/wiki/Profiling_and_Optimizing].

Not Using Version Control

For everything except for short throwaway scripts, or otherwise incredibly short programs, there is no good excuse, not to use a version control system (a.k.a: "revision control systems", "source control systems", or more in general as part of "software configuration management"). This is especially true nowadays given the availability of several powerful, easy to use, open-source (and as a result free-of-charge), and cross-platform, version control systems, that you should have not a lot of problems to deploy, learn and use.

For more information and the motivation behind using version control systems, see the relevant section out of the fifth part of “Perl for Perl Newbies” [http://perl-begin.org/tutorials/perl-for-newbies/part5/#page--version-control--DIR] for more discussion about the motivation behind that, some links and a demonstration.

Some links for further discussion:

- The Better SCM Site [http://better-scm.shlomifish.org/]
- The Free Version Control Systems Appendix of *Producing Open Source Software* [http://producingoss.com/en/vc-systems.html].
- The Wikipedia List of revision control software [http://en.wikipedia.org/wiki/List_of_revision_control_software].
- “You Must Hate Version Control Systems” [http://perlhacks.com/2012/03/you-must-hate-version-control-systems/] - a discussion on Dave Cross’ blog about best practices in the software development industry.

Writing Automated Tests

Automated tests help verify that the code is working correctly, that bugs are not introduced due to refactoring or the addition of new feature, and also provide specifications and interface documentation to the code. As a result, automated tests have been considered a good practise for a long time.

For more information about how to write automated tests, see our page [http://c-begin.wikia.com/wiki/Testing_Tools] about quality assurance in C.

Parsing XML/HTML/JSON/CSV/etc. without a tried-and-tested parser

You should not try to parse HTML, XML, JSON, CSV, and other complex grammars using regular expressions, or worse - using manual character/string tokenisation. Instead, use a tried and tested parsing library, which you should be able to find using a web search.

Generating invalid Markup (of HTML/etc.)

You should make sure that the HTML markup you generate is valid HTML [http://en.wikipedia.org/wiki/XHTML#Valid_XHTML_documents] and that it validates as XHTML 1.0, HTML 4.01, HTML 5.0, or

a different modern standard. For more information, see the “Designing for Compatibility” section [http://www.shlomifish.org/lecture/LAMP/slides/compatibility/] in a previous talk.

Capturing Instead of Clustering in Regular Expressions

If you want to group a certain sub-expression in a regular expression [http://perl-begin.org/topics/regular-expressions/], without the need to capture it (into the `$1`, `$2`, `$3`, etc. variables and related capture variables), then you should cluster them using `(?: … )` instead of capturing them using a plain `( … )`, or alternatively not grouping them at all if it's needed. That is because using a cluster is faster and cleaner and better conveys your intentions than using a capture.

Buffer Overflows

Buffer overflows involve reading or writing after one end of the buffer and can lead to exploitation, or crashes. More information can be found in the Wikipedia article [http://en.wikipedia.org/wiki/Buffer_overflow].

Format String Vulnerabilities (printf/etc.)

When passing a non-literal-constant string as the first parameter to “printf()”/sprintf()” and friends, one runs the risk of format string vulnerabilities [http://en.wikipedia.org/wiki/Uncontrolled_format_string] (more information in the link). As a result, it is important to always use a literal constant string to format the string. E.g:

```
# Bad code


fgets(str,sizeof(str), stdin);
str[sizeof(str)-1] = '\0';
printf(str);
```

should be replaced with:

```
fgets(str,sizeof(str), stdin);
str[sizeof(str)-1] = '\0';
printf("%s", str);
```

One can also use the relevant warning flags [http://stackoverflow.com/questions/9707569/c-array-warning-format-not-a-string-literal] of GCC and compatible compilers to warn and possibly generate an error for that.

Callbacks that do not accept a “void *” context variable

When writing C (and sometimes C++) code, make sure that whenever you accept a function pointer as a callback, that it also accepts a `void *` context variable, which would be passed to the function as well. These variables are sometimes called “batons” because they are passed around. Without this convention, it will be harder to instantiate routines based on this interface, and multi-threading will be harder without the use of thread-local-storage.

So don't do that:

```
# Bad code
```

```
void my_sort(my_type * const array, const size_t count,
    int (*compare)(my_type *, my_type *));
```

And instead do that:

```
void my_sort(my_type * const array, const size_t count,
    int (*compare)(my_type *, my_type *, void *), void * const context);
```

Not Using a Proper Build System

It is a very good idea for C and C++ code to use a good build and configuration system. There's a page listing some prominent alternatives [http://www.shlomifish.org/open-source/resources/software-tools/]. For simple setups, a make file may be suitable, but more complex tasks require a configuration and build system such as CMake [http://en.wikipedia.org/wiki/CMake].

Not Using a Bug Tracker/Issue Tracker

It is important to use a bug tracking [http://en.wikipedia.org/wiki/Bug_tracking_system] system to maintain a list of bugs and issues that need to be fixed in your code, and of features that you'd like to work on. Sometimes, a simple file kept inside the version control system would be enough, but at other times, you should opt for a web-based bug tracker.

For more information, see:

- Joel on Software article about "Painless Bug Tracking" [http://www.joelonsoftware.com/articles/fog0000000029.html]
- "Bug Trackers" list [http://www.shlomifish.org/open-source/resources/software-tools/#bug_tracking] on Shlomi Fish's "Software Construction and Management Tools" page.
- "Top 10 Open Source Bug Tracking Systems" [http://www.thegeekstuff.com/2010/08/bug-tracking-system/]

Sources of This Advice

This is a short list of the sources from which this advice was taken which also contains material for further reading:

1. A large part of this document is derived from a similar document [http://perl-begin.org/tutorials/bad-elements/] written earlier for the Perl programming language.
2. The Book "Perl Best Practices" [http://perl-begin.org/books/advanced/#pbp] by Damian Conway - contains a lot of good advice and food for thought, but sometimes should be deviated from. Also see the "PBP Module Recommendation Commentary" [https://www.socialtext.net/perl5/index.cgi?pbp_module_recommendation_commentary] on the Perl 5 Wiki.
3. "Ancient Perl" [https://www.socialtext.net/perl5/index.cgi?ancient_perl] on the Perl 5 Wiki.
4. chromatic's "Modern Perl" Book and Blog [http://modernperlbooks.com/]
5. The book *Refactoring* by Martin Fowler [http://www.refactoring.com/] - not particularly about Perl, but still useful.

6. The book *The Pragmatic Programmer: From Journeyman to Master* [http://pragprog.com/book/tpp/the-pragmatic-programmer] - also not particularly about Perl, and I found it somewhat disappointing, but it is an informative book.

7. The list “How to tell if a FLOSS project is doomed to FAIL” [https://www.theopensourceway.org/wiki/How_to_tell_if_a_FLOSS_project_is_doomed_to_FAIL].

8. Advice given by people on Freenode's #perl channel [http://perl-begin.org/irc/#freenode], on the Perl Beginners mailing list, and on other Perl forums.

9. Advice given by people on Freenode’s ##programming channel and on other forums.