Guidelines for Writing C Code

Issue 01-bugfix

Martin Becker
Institute for Real-Time Computer Systems (RCS)
Technische Universität München
becker@rcs.ei.tum.de

June 9, 2014

Contents

1 Introduction

This document is a work in progress. It tries to give a number of recommendations, that help to develop safe and unambiguous C programs, in particular for safety-critical systems.

When writing a program, the developer often has a variety of ways to express the same functionality. Some may be "more elegant", others more intuitive to read, and still another way might produce the fastest possible code. However, what we should be concerned with is not primarily the speed of some piece of code, or how good it looks, but rather that *it is correct*.

From the experience, a lot of errors can be avoided by using some basic coding rules, that help to keep an overview over the code, and allow other persons that are interfacing such code to infer its functionality faster. By applying such coding rules, what we get is a code that expresses functionality in a certain way, the *coding style*.

This document suggests a coding style that developed over the years, adopting ideas from fellow students, co-workers as well as established standards, such as *MISRA C* [1]. It does not cover topics related to timing or synchronization, such as threading, mutex deadlocks or similar. It is well-known, that the C language itself does not have sufficient expressiveness to avoid such defects, and for techniques regarding these problems there is much more necessary than only applying coding guidelines. This document does neither account for optimization.

2 Pragmatic Rules

Care for compiler warnings

Resolve all warnings, as far as possible. Each warning may indicate a non-intended behavior that was chosen by the compiler as a consequence of "vague code".

Use static code checkers

Tools such as `cppcheck`[1] can point out a lot of style errors and problems, such as non-initialized variables and exceeding of array bounds.

Use dynamic code checkers

Tools such as `valgrind`[2] can track uninitialized use of variables, errors in dynamic memory management and much more.

Do not use `printf()` for debugging

It seems tempting, but for more than quick occasional checks do not use it. It modifies the execution time and – since output could be buffered – the output is not guaranteed to preserve order and relative timing, in particular with threads. Additionally it takes a lot of effort to place all the `printf`s and remove them thereafter. Use a debugger or introduce a proper logging mechanism instead.

A function should fit on one monitor page

Long functions are tedious to debug and poorly modularized. Break it down into sub-functions, which serves maintainability, automatically enhances documentation and brings a better overview.

Use `doxygen` comments to document code

Doxygen[3] is a tool that can generate an informal documentation of the code from comments that the developer provides. When adding comments of this flavor, it is not only easier to understand the code (since it is documented!), but a HTML or PDF documentation can be generated in seconds.

3 Naming of Variables and Functions

In principle, the developer has all the freedom in choosing names of programming entities. However, this is often where replacing an arbitrarily chosen name with a more succinct and complete one can make a wide difference.

Use hierarchical names

A name/identifier may describe a function that is located inside a logical entity. To make clear to which entity it belongs to, it is suitable to choose identifiers of the form `module_subfunction_function`, e.g., it is immediately clear what the function `file_open_readonly()` is doing.

[1] http://cppcheck.sourceforge.net
[2] http://www.valgrind.org
[3] http://sourceforge.net/projects/doxygen

Add units

For variables, especially as arguments for functions, it is a great help to add the unit as a suffix, e.g., `time_sec`.

Start names of static functions with an underscore

That helps to infer the nature of the function when encountered in code and eases overview.

Avoid numerical terminals

For both maintainability and documentation purposes, do not write code such as `int array[5];`, instead use a preprocessor definition or even better a constant with a descriptive name, to declare the numeric value and then refer to it, e.g., `#define MAX_ELEMS 5` and then `int array[MAX_ELEMS]`.

4 Function Interfaces

Provide return values

Use return values whenever the result (positive or negative) is of interest for the caller. Functions should only return `void` if either no error can occur, no countermeasure or mitigation possible if an error occurs, or not necessary in the sense that the program can continue without degrading.

Use const pointer arguments

Use `const` to indicate clearly which arguments are not changed; the caller can infer which arguments need to be handed over as a copy, or which are used read-only.

Use `static` for internal functions

When a function must only be called from an internal context of a translation unit (e.g., because it has insufficient error checking or is not of any use for the outer world), declare it static to indicate it is "private".

Functions shall not call themselves, either directly or indirectly.

Recursive calling can eventually result in stack overflow.

Functions with no parameters shall be declared with parameter type `void`.

ANSI-C.

If a function returns error information, then that error information shall be tested

The effect and/or return value of a function might be not meaningful or wrong in case the error information is ignored. The program behavior deviates from the specification.

Check preconditions

At the beginning of a function, included checks that the arguments are valid. E.g., check for null pointers or values that are out of expected bounds.

5 Variables

Avoid global variables

Variables that are located outside functions and not static (that is, internal linkage) should be avoided. They introduce side-effects and render the whole program potentially non-re-entrant, i.e., seemingly non-deterministic.

Always initialize

Never declare a variable without initialization.

Minimize scope

Keep the scope of a variable as narrow as possible. That also applies for type definitions.

Avoid implicit casts during assignment

When a type conversion is required, make it explicit by type casting in case the semantics are correct for your application. Else do not cast, but provide an appropriate conversion function.

Write comparisons with constant L-values

When comparing a variable against a constant, always use the constant as an L-value. In case the comparison was accidentally written as an assignment, the compiler will detect an invalid assignment and throw an error.

Avoid static variables in functions

Some systems may not properly initialize such memory, and also functions with such variables are not free of side-effects, causing problems under threading. Rather use contexts[4] that are handed over by the caller.

6 Header Files

Header files shall only contain "public" information

Contents of header files represent information that can be used by any translation unit that `includes` that file. Therefore a header file shall only contain those declarations, `defines` and type definitions, that are necessary to be known by potentially all parts of the program. In contrast, information that is only relevant for one translation unit, shall be placed inside that unit and not in the header, e.g., type definitions required only within one c-file.

Protect against multiple inclusion

To avoid that a header file is included multiple times and causes errors through this, all header files should have the following anatomy:

```
#ifndef HEADER_NAME_H
#define HEADER_NAME_H
/* Body of the header file. */
#endif /* HEADER_NAME_H */
```

[4] Some pointer that the caller gives to the callee as an argument, which can be de-referenced by the callee and contains the necessary data.

There should be no function definitions or variables in header files.

That can cause unintended re-declarations and incarnate global variables.

7 Control Constructs

Always use curly braces

Even for single-statement bodies, use curly braces. A future extension of the code is less likely to introduce errors, and it is easier to read.

Use bounded loops

Apart from exceptional situations, do not use unbounded loops, such as `while(1) { ... }`.

The three expressions of a for loop's header shall only be concerned with loop control.

All `if ... else if` constructs shall be terminated with an `else` clause.

For reasons of logical completeness; unexpected outcomes can at least be detected.

The final clause of a `switch` statement shall be the `default` clause.

For reasons of logical completeness; unexpected outcomes can at least be detected.

All exit paths of a function shall have an explicit return statement with an expression.

Except for functions returning void; otherwise, seemingly non-deterministic return values are possible.

8 Memory Management

Do not use `malloc` or `calloc` after initialization

Each dynamic allocation of heap memory may fail, and potentially take a long time, rendering the execution time and the execution itself seemingly non-deterministic. Instead allocate all memory right after the program starts, and work only with this memory thereafter.

Check return value of malloc

Memory allocation may fail, especially on systems with low memory.

Free memory at exit

All allocated memory must be freed when the program exits in any anticipated way.

9 Built-In tests

Obligatory checks must not be asserts

Checks that are necessary for correct functioning and where there is no appropriate error handling must not have any way to be deactivated during compilation. Since `asserts` can be deactivated, they must not implement essential or vital checks, e.g., preconditions for function parameters.

10 Comments and Documentation

Functions that have prototypes in header files must be documented in the header

This documentation should at least comprise a brief description of the functionality, the possible return values (if applicable), and a brief description of the parameters along with allowed ranges.

Use a file header

At the beginning of each file, use a comment that gives a brief description of the contained functions, author name and date.

Use comment blocks to separate different code blocks

Immense help for readability, e.g.:

```
/*********************
*     VARIABLES      *
*********************/
int x = 0;
double f_Hz = 50;
```

Use TODO and FIXME keywords

Editors such as Eclipse highlight such comments, and tools like Doxygen allow to compile todo lists. Use "TODO" (critical item to be resolved) for things that are not completed or faulty, and "FIXME" (low-prio item to be resolved) for things that are working, but fail under certain boundary conditions or are badly coded.

References

[1] Motor Industry Software Reliability Association. *MISRA-C:2004: Guideliness for the Use of the C Language in Critical Systems*. Mira Books Limited, 2004.

[2] J.J. Labrosse. *Embedded Systems Building Blocks: Complete and Ready-to-Use Modules in C*. R&D Developer Series. Taylor & Francis, 1999.

[3] D.E. Simon. *An Embedded Software Primer*. Number Bd. 1 in An Embedded Software Primer. Addison Wesley, 1999.