



v0.9.1-1  
RELEASE DATE: N/A

---



**TROPOSPHERIC EMISSIONS:  
MONITORING OF POLLUTION (TEMPO)**

**IOC/SDPC C Coding Standards**

October 26, 2020

Prepared by: John E. Davis  
jedavis@cfa.harvard.edu

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Naming Conventions</b>	<b>4</b>
2.1	Variable Names . . . . .	5
2.2	Function Names . . . . .	7
2.3	Typedef Names . . . . .	8
<b>3</b>	<b>Indentation</b>	<b>8</b>
3.1	Indenting Statement Blocks . . . . .	9
3.2	Indenting Statements . . . . .	9
3.3	Indenting Labels . . . . .	9
3.4	Compatibility with other Styles . . . . .	9
<b>4</b>	<b>Functions and Statements</b>	<b>10</b>
4.1	Function Declarations and Prototypes . . . . .	10
4.2	Function Definitions . . . . .	10
4.3	Statements and Flow Control . . . . .	11
<b>5</b>	<b>Comments</b>	<b>12</b>
5.1	Comments in Source Code . . . . .	12
5.2	Comments in Header Files . . . . .	13
5.3	Comments for Specific Statements . . . . .	14
<b>6</b>	<b>Preprocessor Statements</b>	<b>15</b>
<b>7</b>	<b>Structures and Typedefs</b>	<b>16</b>
<b>8</b>	<b>Error Handling</b>	<b>16</b>
8.1	Error Codes . . . . .	16
8.2	Freeing Resources after an Error . . . . .	18
<b>9</b>	<b>Memory Allocation and Resource Management</b>	<b>20</b>
9.1	Allocating Resources . . . . .	20
9.2	Deallocating Resources . . . . .	20
9.3	The malloc function . . . . .	20
<b>10</b>	<b>Signals</b>	<b>21</b>
10.1	The Signal Handler . . . . .	21
10.2	System Calls . . . . .	22
<b>11</b>	<b>Libraries</b>	<b>22</b>
<b>12</b>	<b>Source Code Layout</b>	<b>23</b>
<b>13</b>	<b>Directories and Filenames</b>	<b>23</b>

13.1 Source Code Distribution Layout . . . . .	23
13.2 Filenames . . . . .	24

## 1 Introduction

This document describes the C coding standards for the TEMPO ground systems software. The goal of a coding standard is to produce code that is readable, easy to maintain, and bug-free. Unfortunately, there is no one set of universally adopted coding standards. The standards outlined in this document are a result of years of experience in developing, maintaining, and studying source code.

In this document, the coding standards fall into one of two classes: rules that are mandatory, and rules that are guidelines or suggestions. A rationale is provided for all of the mandatory rules, and for most of the guidelines.

Since C++ may be regarded as a superset of C, many of the rules given in the document also pertain to C++. However, they are different languages and some of the rules that are appropriate to C are not for C++.

## 2 Naming Conventions

The main objective of a coding standard is to provide a set of rules that are designed to help software developer produce code that is easy to maintain. But doing so requires that the code also be understandable. If a new person is brought into a project to maintain or enhance a particular piece of code, it is unrealistic to expect this person to immediately understand the code as a whole no matter what standard has been followed. However, it is reasonable to assume that the new person should be able to understand the code on a smaller scale such as a function or a code block. The naming conventions described in this section are designed to allow a developer to grasp the code at this more basic level.

To better illustrate how a naming scheme can facilitate such an understanding, consider the following code fragment using a lower camelCase naming convention:

```
if (articleLines == NULL)
    articleLines = lineNumber;
else
{
    lineNumber->next = currentLine->next;
    lineNumber->prev = currentLine;
    currentLine->next = lineNumber;

    if (lineNode->next != NULL) lineNode->next->prev = lineNumber;
}
currentLine = lineNumber;
```

It should be obvious that this code involves adding an item to a linked list that is implemented using pointers. What else can be said about this code? Can anything be said about the state of the program after the first assignment? Suppose that it is decided that using an array instead of a linked list will result in cleaner and faster code in the function where this occurs. How much work would be involved in making such a change? The

answer to both of these questions depends upon the scope of the variables involved, but the above code fragment as written says nothing about the scope. The naming scheme given in this section is designed to provide such information.

## 2.1 Variable Names

Both C and C++ are case-sensitive languages. This case-sensitivity is exploited to achieve the primary goal set out in the introduction. We distinguish variables according to their scope. For C, there are essentially four types of variables:

**Local** Variables that are local to a function or method.

**Static** Variables whose scope is limited to the file or module where they are defined.

**Global** Variables that may be referenced by name throughout the program (global scope).

**Macros** Variables associated with the preprocessor

In the above, no distinction is made between an automatic (stack allocated) local variable and one statically allocated to a function. The scope of both of these variables is limited to the function where they are defined. The language also permits block-local variables, but these will also be lumped into the general category of local variables. The scoping rules for C++ are more complicated and will be discussed in the appendix.

Throughout this document, the term *non-local* is used to indicate a scope that is wider than function-scope.

**2.1.0.** The use of non-local variables discouraged. [\[Guideline\]](#)

*Rationale:* Code containing non-local variables can be difficult to understand due to the scope of these variables. Their presence is also problematic for threaded and reentrant code.

**2.1.1.** Local variables shall be all lowercase. [\[Mandatory\]](#)

**2.1.2.** Static variables shall be fully capitalized<sup>1</sup> [\[Mandatory\]](#)

**2.1.3.** Global variables shall be fully capitalized and share a common prefix. [\[Mandatory\]](#)

**2.1.4.** Macro names shall be all uppercase [\[Mandatory\]](#).

*Rationale:* These rules require that an association be made between the scope of a variable and how its name is composed of upper and lower case characters. If a line of code contains a variable that is capitalized, then it is immediately obvious that execution of that line of code is affected by, or can affect, the state of the program outside the function containing the line. This naming scheme also prevents problems associated with local variables shadowing non-local ones.

**2.1.5.** Block local variables shall not have the same name as other local variables in the function where they are defined. [\[Mandatory\]](#)

---

<sup>1</sup>By “fully capitalized”, it is meant that the name start with a capital letter as well as any character that follows an underscore character. For example, `Variable_Name` is permitted, but `Variable_name` is not.

*Rationale:* The “shadowing” of a local variable by a block variable of the same name makes the code harder to understand since a single name refers to different variables in the same function.

**2.1.6.** Upper CamelCase names are discouraged. [\[Guideline\]](#)

*Rationale:* Uppercase characters may only be used in the names of non-local variables. If the variable is important enough to have such scope, then it is important enough to use an underscore character to delineate the words that make up the name. (Note that a lower camelCase name is forbidden by the mandatory rules governing variable names.)

**2.1.7.** If the code is part of a software library, then the prefix for global variables that are exposed as part of the library’s API must be different from the prefix used for global variables that are not part of the API [\[Mandatory\]](#).

*Rationale:* Whenever an attempt is made to clean-up or simplify code, eliminating global variables should be part of that process. Or perhaps a global variable may no longer be necessary if part of the code is changed to use a different algorithm. For both of these scenarios it is important to know if the global variable is part of the API, since such a variable cannot be eliminated without violating the API.

**2.1.8.** Non-local variables shall be given names that describe what they represent. [\[Mandatory\]](#)

*Rationale:* If a variable is deemed important enough to be non-local then the name of the variable should reflect that importance.

## Example

Consider once again the earlier code fragment but this time with the variables renamed to follow the naming conventions mandated above:

```
if (Slrn_Article_Lines == NULL)
    Slrn_Article_Lines = node;
else
    {
        node->next = Current_Line->next;
        node->prev = Current_Line;
        Current_Line->next = node;

        if (node->next != NULL) node->next->prev = node;
    }
Current_Line = node;
```

This code fragment was extracted from the source code for the **slrn** newsreader, which uses `Slrn_` as the prefix for global variables. With this naming scheme, it is immediately obvious that:

- `node` is a local variable.

- `Slrn_Article_Lines` is a global variable.
- `Current_Line` is a static variable with file scope.

This means that the assignment

```
Slrn_Article_Lines = node;
```

has global consequences– it affects the global state of the program. The assignment

```
Current_Line = node;
```

also has non-local consequences but its effects are confined to the current file. From this it is easy to see that if the algorithm is changed to eliminate `Current_Line`, the change would only involve the file containing this variable. Any changes to the algorithm that affect `Slrn_Article_Lines` could be a major effort since the scope of this variable is global.

It is also apparent from the above code that all of the variables are pointers. Any coding standard that utilizes a naming scheme that emphasizes the type of a variable is misguided since one can usually glean something about a variable’s data type just by looking at how the variable is used.

## 2.2 Function Names

Like variables, functions are also distinguished by their scope. For the purposes of this document, functions fall into 2 classes: those that are private to a file, e.g.,

```
static int private_function (void) {...}
```

and those that are global (lack the `static` attribute).

**2.2.0.** All global function names shall share a common prefix. **[Mandatory]**

*Rationale:* The common prefix not only serves to distinguish global functions from private ones, but it also helps prevent name clashes with symbols in external libraries. Moreover, if the code is re-factored to create a separate library, the prefix can serve as a guide to what should go into the library.

**2.2.1.** If the global function is part of a library, then the prefix given to the functions that are part of the library’s public API must be different from the prefix given to those global functions that are not exposed by the API. **[Mandatory]**

*Rationale:* Being able to distinguish between those functions that are exposed by the API from those that are not can greatly facilitate refactoring.

**2.2.2.** With the exception of the prefix on global function names, all function names shall be in lowercase **[Mandatory]**.

*Rationale:* This convention avoids name clashes with non-local variables since such variables must be “fully” capitalized.

## 2.3 Typedef Names

The typedef statement is used in C to declare a new datatype. For example,

```
typedef struct
{
    double x, y;
}
Point_Type;
```

declares a new datatype called `Point_Type` that represents a pair of double precision floating point numbers. Like variables, the type name has a scope, and will follow the same naming convention as for variables.

**2.3.0.** A datatype name with scope that is local to a file shall be fully capitalized.

[Mandatory]

**2.3.1.** A datatype name with global scope shall be fully capitalized and use the same common prefix as for global variables. [Mandatory]

**2.3.2.** If the datatype is global but not part of a library's API then its prefix shall be different from the API prefix. [Mandatory]

*Rationale:* Understanding at a glance the scope of the datatype makes code more readable and maintainable.

Many of the standard C datatypes, such as `size_t` use the convention that the type name ends with the suffix `_t`. This convention will not be followed.

**2.3.3.** The name of a datatype shall use the suffix `_Type`. [Guideline]

*Rationale:* Using the `_Type` suffix clearly indicates that the name is referring to a datatype. In addition, the POSIX standard reserves datatype names that end with `_t`.

## 3 Indentation

Properly indented source code is important for visually conveying the structure and flow control of the code. There are a number of different indentation styles in common use. These styles generally fall into two categories: those that have the opening brace of a code block on the same line as the control statement (e.g., K&R style), and those where the opening brace is alone on the line that follows the control structure (e.g., Allman style).

Indenting a line of code involves inserting whitespace consisting of some combination of TABs or SPACES. TABs present a problem because the amount of whitespace associated with a TAB varies. Some programmers set their editor to use 2 column TABs, some use 4 column, some use 8 column, etc. For this reason, using a TAB character to create a specific amount of indentation is not possible in the absence of an agreed upon TAB setting.

**3.0.0.** Indentation whitespace shall consist only of SPACE characters. [Mandatory]

*Rationale:* The displayed width of a TAB character can vary.



## 3.1 Indenting Statement Blocks

**3.1.0.** The opening and closing braces of a code block shall be on lines by themselves.

[Mandatory]

**3.1.1.** The opening brace shall be indented to at least the level of its control

statement. [Mandatory]

**3.1.2.** The closing brace shall be vertically aligned with its matching brace [Mandatory].

*Rationale:* Code that has the opening brace on the same line as the control statement has a more cramped look than code that has the brace on the line following the control statement. Moreover, the latter style more clearly delineates the control statement and the block that follows it.

## 3.2 Indenting Statements

**3.2.0.** Statements contained in a block shall be indented at least two spaces to the right of the brace defining the block. [Mandatory]

**3.2.1.** It is recommended that 3 spaces be used for the indent amount. [Guideline]

*Rationale:* If the code were allowed to be at the same level as the braces, it would not be indented at all, which would defeat the purpose of indenting (this requirement could also read that the code shall be indented). Using at least two spaces makes the indent more noticeable.

## 3.3 Indenting Labels

There are two types of labels considered here: labels that denote the target of a `goto` statement, and `case` labels in a `switch` statement.

**3.3.0.** Labels that are the target of a `goto` statements must not be indented. That is, such a label must begin in the first column. [Mandatory]

*Rationale:* Putting the label in the first column serves as a visual warning that the function involves non-local flow-control.

**3.3.1.** Each `case` label shall be out-indented from the code that it labels, but indented to at least the level of the containing brace of the `switch` block. [Mandatory]

*Rationale:* Out-indenting the `case` label makes it easier to identify the labeled sections of the `switch` statement. The `case` statement should be indented to at least the level of the `switch` since it is subordinate to the `switch` statement.

## 3.4 Compatibility with other Styles

The indentation rules dictated above are consistent with the following well-known indentation styles: Allman, GNU, and Whitesmith/Wishart. Preference is given to a variant of the GNU style where statements are indented by 3 columns inside the surrounding braces. The K&R-like styles are not permitted.

## 4 Functions and Statements

### 4.1 Function Declarations and Prototypes

- 4.1.0. A function that is used only in the file where it is defined shall be declared as `static` [\[Mandatory\]](#)

*Rationale:* In addition to minimizing name collisions and helping to reduce the size of the global namespace, `static` functions may also perform faster than non-`static` equivalents. For example, the optimizer may be able to integrate them into the caller, and the compiler would not have to generate position-independent code (pic) that would require extra dereferences during runtime. Moreover, such functions are easier to maintain since nothing outside the file explicitly depends upon them.

- 4.1.1. Functions that are non-`static` shall be explicitly declared as `extern` in a header file associated with the file containing the function definition. [\[Mandatory\]](#)
- 4.1.2. The function prototype defined in a header file shall be written in the ANSI form, which contains a list of parameters and types. [\[Mandatory\]](#)
- 4.1.3. The file containing the function definition shall include the header file containing the function prototype. [\[Mandatory\]](#)

*Rationale:* This is to ensure consistency with the definition of the function and its declaration. ANSI prototypes allow the compiler to check that the function is called with the proper number of arguments and types.

- 4.1.4. If a function takes no arguments, then it must be declared with `void` in the argument list of the function definition and its prototype. [\[Mandatory\]](#)

*Rationale:* Using `void` explicitly indicates that the function takes no arguments.

- 4.1.5. No function shall call another function whose prototype is not available to the caller. [\[Mandatory\]](#)

*Rationale:* There is no guarantee that the external function will be called properly without a prototype available to the caller.

### 4.2 Function Definitions

- 4.2.0. Avoid writing functions that return nothing (`void`) unless there is no conceivable way that the function can fail, either now or in a future version of the function. [\[Guideline\]](#)

*Rationale:* Most functions can fail and the return value could be used to indicate if it succeeded or failed. See section 8 on error handling for more information.

- 4.2.1. Avoid creating functions that contain a relatively large number of parameters. [\[Guideline\]](#)

*Rationale:* Functions that contain a large number of parameters can be unwieldy to maintain. The presence of a large number of parameters usually indicates that the function was poorly conceived. If a number of the parameters are related,

then the related parameters should be placed in a typedefed object and a pointer to the object passed to the function.

**4.2.2.** Try to keep the body of a function relatively short. [\[Guideline\]](#)

*Rationale:* Functions should have a well-defined purpose. A function that consists of many lines of code is probably trying to do too much. Such a function should be broken into smaller, more manageable pieces, which may lend themselves to code-reuse.

**4.2.3.** Do not use the `register` attribute. [\[Mandatory\]](#)

**4.2.4.** The compiler is most likely better at determining which variables should be kept in a register. [\[Guideline\]](#)

### 4.3 Statements and Flow Control

**4.3.0.** A `return` statement should be used at any point where the control logic dictates that the function should return. [\[Guideline\]](#)

*Rationale:* Except for the simplest of functions, a function may contain many possible paths through it. Longer paths are usually more difficult to understand than shorter ones. A person studying the code will have to explore these various paths and the sooner a path leads to a `return` statement, the shorter that path will be.

Many coding standards say that each function shall have only one `return` statement. To meet this constraint, the developer would have to introduce extra control variables, making the code even more difficult to understand.

**4.3.1.** Use the `break` statement at the point where it is necessary to break out of a looping construct. [\[Guideline\]](#)

**4.3.2.** Use the `continue` statement at the point where it is necessary to continue the next iteration of the looping construct. [\[Guideline\]](#)

*Rationale:* The rationale for these is the same as for the `return` statement. Not using these statements may require extra code, which would make the code less understandable.

**4.3.3.** The use of the `goto` statement should be avoided. [\[Guideline\]](#)

*Rationale:* A `goto` statement permits non-local flow and make code difficult to understand. A `goto` statement should only be used for error handling (see section 8) or when it actually adds clarity to the code.

**4.3.4.** The use of `longjmp/siglongjmp` should be avoided [\[Guideline\]](#)

*Rationale:* The use of these non-local control functions are rarely necessary and make a program difficult to understand and maintain.

**4.3.5.** A `goto` statement shall only jump to a target in a block containing the `goto` statement [\[Mandatory\]](#).

*Rationale:* Jumping to a target in an inner block introduces code maintenance issues and can jump over initializations for the block.

**4.3.6.** A switch statement must always have a `default` label. [\[Mandatory\]](#)

*Rationale:* The presence of the `default` label ensures that all values of the `switch` control variable are handled. This also provides a convenient place to test for invalid values.

**4.3.7.** When comparing a pointer to `NULL`, the pointer shall be explicitly tested against `NULL` and not `0`. [\[Mandatory\]](#)

*Rationale:* Comparing the pointer variable to `NULL` and not `0` is a visual indication that the variable is a pointer. Remember, the more visual cues that can be provided about the code, the more understandable the code will be.

## 5 Comments

Comments are used in source code for many purposes, including: to describe the implementation of an algorithm, to document how a function or variable is to be used, and for metadata such as the copyright, developer name, etc.

**5.0.0.** C++ style comments (`//`) shall not be used in C code. [\[Mandatory\]](#)

*Rationale:* C and C++ are different languages. Although some compilers accept C++ style comments, they are not part of the C language.

**5.0.1.** Inside a comment, the string `/*` shall not appear. [\[Mandatory\]](#)

*Rationale:* Such a string appearing in a comment would most likely be a result of commenting out code that contains comments. However, neither C nor C++ permits nested comments.

**5.0.2.** Comment only what really needs to be commented, and do not comment anything that is obvious from the code. [\[Guideline\]](#)

*Rationale:* Comments should add value to the code, and commenting the obvious adds no value. Experience shows that comments and code can get out of sync such that the comments no longer describe what the code does. When faced with this inconsistency, should one believe the comment and conclude that the code is flawed? No comments are better than incorrect comments.

### 5.1 Comments in Source Code

**5.1.0.** Each non-`static` function shall contain a comment describing what it does, how it is to be used, and the return value. [\[Mandatory\]](#)

**5.1.1.** If the function is a member of the public API of a library, then the documentation should be put in the public header file, otherwise, it should appear in the source code where the function is implemented. [\[Guideline\]](#)

**5.1.2.** If the algorithm was derived from an external source, the comment shall contain a reference to the source. [\[Mandatory\]](#)

- 5.1.3.** The documentation for the non-`static` functions should be written in a way that permits them to be easily extracted by an external program, e.g., `doxygen`. [\[Guideline\]](#)

*Rationale:* Such a comment will give the developer an overview of the algorithmic details of the function making it easier to understand. If the algorithm is described in more detail elsewhere, having a reference to the source can be beneficial.

- 5.1.4.** If the implementation involves some subtle logic that is not readily apparent to someone not familiar with the code, then that part of the algorithm shall contain a comment describing the subtlety. [\[Mandatory\]](#)

*Rationale:* For a better understanding of the code, anything that is not obvious should contain a comment.

- 5.1.5.** Do not waste time commenting every single local variable. [\[Guideline\]](#)

*Rationale:* Some coding standards dictate that comments be written to describe every single local variable. This practice is rarely useful and just makes the code look denser than it is. Instead, give the variables meaningful names. If all local variables need to be documented to understand the function, then the function should be rewritten or broken into smaller, more manageable pieces.

## 5.2 Comments in Header Files

- 5.2.0.** A usage comment shall be given for each (`extern`) function or variable in a header file. [\[Mandatory\]](#)

- 5.2.1.** If the function returns an allocated resource, the comment shall indicate how the object is to be deallocated. [\[Mandatory\]](#)

- 5.2.2.** The comment shall indicate the nature of the return value, i.e., whether or not it is an error code. [\[Mandatory\]](#)

- 5.2.3.** If the header file represents the public API of a library, the comment should include enough information for a developer to use the function without having access to the source code. [\[Guideline\]](#)

*Rationale:* The header file provides a natural place for a developer to find concise usage information about functions and variables. If the function is part of a library, the developer may not have access to the library's source code. In such a case, the header file should include information sufficient for the developer to use the API without the source.

- 5.2.4.** Every declared variable in a header file shall have a brief comment describing it. [\[Mandatory\]](#)

- 5.2.5.** If a structure definition appears in a header file, then every field of the structure shall include a brief comment describing it. [\[Mandatory\]](#)

- 5.2.6.** If the header file represents the public API of a library, the comment should include enough information for a developer to use the variable without having access to the source code. [\[Guideline\]](#)

*Rationale:* To ensure that global variables and structures are used properly, they should be documented. The header file is a natural place for the developer to look for a concise description of these objects.

## 5.3 Comments for Specific Statements

**5.3.0.** In a `switch` statement, if control flows from a non-empty code block of one `case` label to the next without a `break` statement, then a comment such as `/* fall through */` shall be placed on a line by itself where the `break` has been omitted. [\[Mandatory\]](#)

*Rationale:* This comment indicates that the `break` statement was purposely omitted.

An example of the previous rule is:

```
switch (argc)
{
    default:
        myprog_set_error (MYPROG_ILLEGAL_USAGE);
        return -1;
    case 2:
        file2 = argv[2];
        /* fall through */
    case 1:
        file1 = argv[1];
        break;
    case 0:
        no_files = 1;
        break;
}
```

**5.3.1.** If the line of code immediately preceding a label statement is not a `return` statement, then a comment such as `/*fall through*/` shall be placed on the line before the label. [\[Mandatory\]](#)

*Rationale:* This coding standards document strongly suggests that the `goto` statement be rarely, if ever, used for anything but error handling. Hence, the label is most likely the target of a `goto` statement that executes when an error occurs. The `/*fall through*/` comment indicates that the direct path from above is an intentional one and is not simply due to a missing return statement.

**5.3.2.** A looping structure with an empty body should include a comment such as `/*empty*/` in the body. [\[Guideline\]](#)

*Rationale:* Such a comment indicates that the body was purposely left empty and not something for future expansion.

## 6 Preprocessor Statements

**6.0.0.** The preprocessor designation character (`#`) should be placed in the first column and never indented. [\[Guideline\]](#)

*Rationale:* Doing so makes the preprocessor statements more visible. The text following the `#` character may be indented.

**6.0.1.** Preprocessor statements shall be indented by one or more spaces inside a conditional preprocessor statement. [\[Mandatory\]](#)

*Rationale:* Preprocessor statements are easier to understand when indented, e.g.,

```
#ifdef HAVE_FOO
#  define local_foo foo
#  if defined(bar)
#    define HAVE_BAR 1
#  endif
#else
#  define HAVE_BAR 0
#endif
```

**6.0.2.** Macro names shall not contain lowercase characters. [\[Mandatory\]](#)

*Rationale:* This rule reflects common usage and makes it easy to distinguish a macro from a variable or function.

**6.0.3.** Macros appearing in a header file should be unique and descriptive. [\[Guideline\]](#)

**6.0.4.** When redefining a macro, first undefine it using `#undef`. [\[Mandatory\]](#)

*Rationale:* Uniqueness guarantees that an existing macro will not get inadvertently redefined. Descriptive helps ensure uniqueness and makes the code using the macro more comprehensible. Explicitly using `#undef` to undefine a macro before redefining it indicates that that the macro was purposely redefined.

**6.0.5.** Macros should have no side-effects [\[Guideline\]](#)

**6.0.6.** If the side-effect of a macro is to modify a variable, then it shall be documented as doing so. [\[Mandatory\]](#)

*Rationale:* A macro with side effects makes the code harder to follow by concealing such changes.

**6.0.7.** If a macro takes arguments, then each of the macro's dummy arguments shall be enclosed in parenthesis when used in the body of the macro. [\[Mandatory\]](#)

**6.0.8.** Try to write macros that evaluate its arguments only once. [\[Guideline\]](#)

*Rationale:* The parenthesis avoids pitfalls that could arise when compound expressions as passed as an argument to the macro.

**6.0.9.** If the macro introduces new local variables, those local variables shall be given names that end in an underscore. [\[Mandatory\]](#)

*Rationale:* Using a local macro variable with a name such as `x_` helps avoid collisions with other variables.

## 7 Structures and Typedefs

**7.0.0.** If a given structure is to be used in several places, then it should be given a new type via the `typedef` statement. [\[Guideline\]](#)

*Rationale:* If the structure is used often, then it is important and should be elevated to a new datatype. Using the new type-name in declarations is simpler than the alternative requiring the addition of the `struct` keyword.

**7.0.1.** Try to confine code that deals with the internals of a structure to a single file. If this is possible, then make the public form of the structure an opaque datatype. [\[Guideline\]](#)

*Rationale:* Making the public data type opaque allows the actual implementation of the type to be changed without affecting code outside the file where it is implemented. This is particularly good practice for libraries.

**7.0.2.** Avoid passing structures (or structured-types) to functions by value. Pass them by reference. [\[Guideline\]](#)

*Rationale:* Passing by value involves copying the structure, whereas passing it by reference uses just the address.

**7.0.3.** Do not create a typedef whose intent is to conceal pointer usage of the underlying type. [\[Mandatory\]](#)

*Rationale:* Pointers are part of the language and trying to conceal this fact simply obfuscates the code.

As an example, consider this piece of code:

```
int bad (Bad_Type b)
{
    return b->x + b->y;
}
```

From the usage, it is clear that `b` must be a pointer valued variable. However, its declaration totally obscures this fact.

## 8 Error Handling

Perhaps that the biggest mistake an inexperienced programmer makes is insufficient error handling. Lack of proper error handling results in code that crashes and software that is very difficult to maintain.

### 8.1 Error Codes

**8.1.0.** If it is possible for a function to fail, then it shall provide a mechanism for the caller to know that it failed and why it failed. [\[Mandatory\]](#)

*Rationale:* Such functions will eventually fail if called enough times. The caller needs to know the nature of the failure so it can take the appropriate action.



There are many error handling methodologies in use. The mechanism described in this section is perhaps the most commonly used in C, and has stood the test of time: the use of an integer status code where a negative value represents an error, and a non-negative value represents success. In the following, the term error code will be used to refer to the negative values of the status code. Implementation of this paradigm involves enumerating the possible integer values of the error code, and a mapping from the integer error code to a string that describes the error. Note that this does not involve enumerating the various levels of success. One can do so, but it is unnecessary.

**8.1.1.** Error codes shall be represented using an `int` or an `enum` [Mandatory]

*Rationale:* Both representations have advantages and disadvantages. For example, an `enum`'s range of values cannot be extended during runtime, which is something that some applications may want to do. The disadvantage of using an `int` is that the symbolic constants enumerating the possible values would be represented by preprocessor symbols. Either representation is permitted with a slight preference given to using an `int`.

**8.1.2.** There shall be a mechanism to map an error code to a human readable text string that describes the error. [Mandatory]

*Rationale:* It is better to display an informative message to the user than simply the value of an error code.

For functions that return pointers, it is natural to have the function return a `NULL` value to signify an error. But the `NULL` value alone does not indicate the nature of the error. One solution in this case is to use a non-local variable that contains the value of the error code. The same issues that apply to other non-local variables also apply to this one, e.g., issues with thread safety, etc.

**8.1.3.** The code should use a non-local variable that tracks the error state of the program. [Guideline]

**8.1.4.** If a non-local variable is used to track the error state, then its value shall only be obtained or set using accessor functions. [Mandatory]

**8.1.5.** The non-local variable shall be declared as static. [Mandatory]

*Rationale:* Accessor functions add a great deal of flexibility that the variable alone cannot provide. For example, when such a function is called to set the error code, a message could be written to a log file. If multiple threads are involved, the functions would contain any mutex-specific code. Also these functions could be used to implement thread-specific error handlers.

Except for trivial cases, most functions can fail in one way or another. If a function calls other functions, those functions could fail, causing the caller to be in an error state. Even if the function being called is documented as one that does not fail, that does not mean this will be true in subsequent versions of the code. It seems unwise to assume that a function that has no conceivable way of failing now will not do so in a future version. For this reason, almost all functions should return an error code, even if that code is 0 (success), e.g.,

```
int function (int arg)
{
    some_function_that_returns_void (arg);
    return 0; /* success */
}
```

**8.1.6.** Almost all functions should return a status code. Functions should rarely return void. [\[Guideline\]](#)

*Rationale:* It is very difficult to add support for a function that went from not returning an error code in one version to one returning an error code in the next version. This would require code modifications where the function is called, possibly changing what the caller returns, which would propagate up the call chain. It is better add the error handling code at the outset instead of waiting until it is too late.

**8.1.7.** If a function returns an error code, then the returned value must be handled in some way by the caller. [\[Mandatory\]](#)

**8.1.8.** If a function returns an error, but the caller does not handle the error, then the return value of the function shall be cast to void [\[Mandatory\]](#).

*Rationale:* Since the function is returning an error code, it can fail and that failure must be dealt with by the caller. If it has been determined that the failure can be ignored by the caller, then the caller must cast the return value to void to provide a visible reminder that a possible failure is going unchecked, e.g.,

```
(void) function_that_returns_an_error_code ();
```

## 8.2 Freeing Resources after an Error

When a function experiences an error, often there is no simple way for the function to recover. As a result, the function will most likely simply return the error code to the caller for resolution. If the function experiencing the error has allocated any resources, those resources should be deallocated prior to returning. The method advocated here for doing this is to use a simple `goto` statement that transfers control to the section of code responsible for the clean-up. Using the `goto` statement for this purpose is more likely to result in more readable code than a `goto`-less alternative.

As an example, consider the following function involving the allocation of several resources:

```
int some_function (void)
{
    ResourceA_Type *a;
    ResourceB_Type *b;
    ResourceC_Type *c;
    int status;

    status = new_resource_a (&a);
    if (status < 0) return status;
    status = new_resource_b (&b);
```

```
    if (status < 0)
    {
        delete_resource_a (a);
        return status;
    }
    status = new_resource_c (&c);
    if (status < 0)
    {
        delete_resource_b (b);
        delete_resource_a (a);
        return status;
    }
    /* code that uses the resources here */
    delete_resource_c(c);
    delete_resource_b(b);
    delete_resource_a(a);
    return status;
}
```

Now consider the same code written to use a goto:

```
int some_function (void)
{
    ResourceA_Type *a = NULL;
    ResourceB_Type *b = NULL;
    ResourceC_Type *c = NULL;
    int status;

    status = new_resource_a (&a);
    if (status < 0)
        goto return_status;
    status = new_resource_b (&b);
    if (status < 0)
        goto return_status;
    status = new_resource_c (&c);
    if (status < 0)
        goto return_status;
    /* code that uses the resources here */
return_status:
    if (c != NULL) delete_resource_c(c);
    if (b != NULL) delete_resource_b(b);
    if (a != NULL) delete_resource_a(a);
    return status;
}
```

Not only is the latter code a bit shorter, it is also more maintainable in the sense that the

resource deallocation is contained in one place and not distributed throughout the code.

## 9 Memory Allocation and Resource Management

Most non-trivial programs involve memory allocation or some sort of dynamically allocated resources. Examples of functions that create such resources include `malloc` and `fopen`. The rules presented in this section are designed with the goals of making the code more robust and free of memory leaks.

### 9.1 Allocating Resources

- 9.1.0. If a resource is dynamically allocated, then the allocation must be checked for failure. **[Mandatory]**

*Rationale:* Many novice programmers fail to check the return value from functions such as `malloc`. While such functions may rarely fail for small allocations, when they do (and they eventually will), the program will crash if not properly checked.

### 9.2 Deallocating Resources

- 9.2.0. Any dynamically allocated resource that is attached to a local variable and not returned by the function shall be deallocated before returning. **[Mandatory]**
- 9.2.1. If a non-local variable is assigned to a dynamically allocated resource, it must be deallocated before assigning a new resource to it. **[Mandatory]**
- 9.2.2. If it is possible for the dynamically allocated resource attached to a non-local variable to not be properly deallocated by the OS upon program exit, then the program shall properly deallocate the resource. **[Mandatory]**

*Rationale:* Not freeing the memory will result in a memory leak. Not properly closing a file could result in data loss.

### 9.3 The `malloc` function

The `malloc` function is used to dynamically allocate a specified number of bytes. The number of bytes needed by an algorithm will vary and occasionally that number could be zero. For example, it is perfectly reasonable for an array of objects to contain no objects. What does `malloc` return when asked to allocate 0 bytes? The C standard says that it may return `NULL` or a unique pointer that can be successfully passed to `free`. The problem with this ambiguity is that `NULL` also represents failure. Did it return `NULL` because that is always what it returns when allocating 0 bytes, or did it return `NULL` because it failed to find a unique pointer that could be passed to `free`?

To deal with this issue, it is recommended that wrappers for `malloc`, `realloc`, and `calloc` be used. This is also recommended because it fits in with the error handling guidelines given in section 8. An example of such a wrapper for `malloc` is

```

void *my_malloc (size_t size)
{
    void *ptr;
    if (size == 0) size = 1;
    if (NULL == (ptr = malloc (size)))
        my_set_error (MY_MALLOC_ERROR);
    return ptr;
}

```

## 10 Signals

A Unix signal is an asynchronous message sent to a process to tell it that some event occurred. For example, pressing Ctrl-C while running a program may result in a signal (SIGINT) being sent to the process indicating that it should interrupt what it is doing and terminate. A program can establish a signal handler that will automatically be called when a signal arrives. When a signal is delivered, the program will stop what it is doing, run the signal handler, and then resume where it left off prior to the signal. Needless to say, signal handling can be complicated and is fraught with race conditions making the code difficult to debug.

### 10.1 The Signal Handler

All signals have a default signal handler that will perform a signal-specific action. For example, the default handler for SIGINT is to terminate the program. Most signals permit the use of a program specified signal handler. Some signals (SIGSTOP and SIGKILL) do not permit changing the default handler.

Traditionally a handler was established using the `signal` function. Unfortunately, the semantics associated with this vary with the OS. To overcome this issue, POSIX specifies an alternative called `sigaction`.

**10.1.0.** Signal Handlers shall be established using the POSIX `sigaction` function.

[Mandatory]

*Rationale:* This function allows control over the semantics of the signal handler.

Since signals are asynchronous events, a signal can arrive at any point in a calculation. For example, one could arrive while changing a pointer in a linked list. In such a case, while the signal handler is running, the list would be in an undefined or incomplete state. As such, the linked list should not be accessed while the signal handler is executing. In fact, since the signal could arrive anytime, it is unwise for the signal handler to do anything more than set a non-local variable to indicate that the handler was invoked. Although there are some signal-safe functions that may be called asynchronously, most cannot.

**10.1.1.** The signal handler shall only call functions that are known to be signal-safe.

[Mandatory]

- 10.1.2.** It is simplest to just set a non-local variable, and then reinstate the signal handler if needed. [\[Guideline\]](#)
- 10.1.3.** The only non-local variables that a signal handler shall set are those declared as `volatile` [\[Mandatory\]](#).

*Rationale:* Calling a function that is not signal-safe will lead to undefined behavior. The `volatile` keyword is a hint to the compiler that the value of the variable could change anytime.

## 10.2 System Calls

If your program is using low-level I/O routines such as `read`, `write`, or `select`, then it is also probably handling signals. When a system call gets interrupted by a signal, it may return a failure code of -1 with the `errno` variable set to `EINTR` indicating that it was interrupted by a signal. The program should be prepared to restart the system call, e.g.,

```
while (-1 == (nread = read (fd, buf, count)))
{
    if (errno == EINTR)
        continue;
    .
    .
}
```

- 10.2.0.** The caller shall check for `errno == EINTR` from a failed system call and take the appropriate action. [\[Mandatory\]](#)

*Rationale:* It cannot always be assumed that system calls are automatically restarted. Hence, this is a valid failure mode that must be handled.

- 10.2.1.** Consider adding support for a callback function to the `EINTR` block. This function can check what signal handlers have been triggered and carry out any tasks associated with them. [\[Guideline\]](#)

*Rationale:* This is one way of dealing with signal handlers that simply set a non-local variable to indicate that a signal was received.

## 11 Libraries

- 11.0.0.** When developing code for a library, under no circumstances shall the new code terminate the program if it failed to acquire a resource. [\[Mandatory\]](#)

*Rationale:* Only the application using the library should terminate itself. Instead, if resource allocation fails, the function shall return an error code.

- 11.0.1.** The most experienced developers on a team should be creating the libraries. The junior developers should be creating the apps that utilize the library. [\[Guideline\]](#)

*Rationale:* A library is a reusable software component that is more difficult to create than an app. For example, the library should be binary compatible from one minor

release to the next. Also, the API should not expose any of the specific implementation details of the library. Producing such a library that meets these goals takes experience.

- 11.0.2.** It is a good idea to perform basic argument checking in the functions that are in the public API. [\[Guideline\]](#)

*Rationale:* It is better to program defensively because a library developer has little control over the quality of the app using the library.

## 12 Source Code Layout

The recommended structure for source code files is as follows:

```
/* Single line summarizing the code in the file. */
/* Copyright */
/* include for local config header */
/* includes for system header files */
/* includes for local header files */
/* typedefs, structures, and variables */
/* functions */
```

The recommended structure for header files is:

```
#ifndef UNIQUE_IDENTIFIER_H
#define UNIQUE_IDENTIFIER_H
/* Single line describing the intent of the header file */
/* Copyright */
/* declarations and prototypes */
#endif /* UNIQUE_IDENTIFIER_H */
```

Here the preprocessor symbol `UNIQUE_IDENTIFIER_H` is a unique preprocessor identifier that is meant to avoid multiple inclusions of the file.

- 12.0.0.** Each file should contain a copyright notice. [\[Guideline\]](#)

*Rationale:* The copyright information allows the owner of the copyright the right to dictate who can copy the work, and under what conditions. This protects both the copyright holder and the user of the code.

## 13 Directories and Filenames

### 13.1 Source Code Distribution Layout

The top-level directory is the most important. It should contain a minimum of files and top-level subdirectories so that important files such as `README` or `INSTALL` are readily visible.

- 13.1.0.** Source code should *not* be placed in the top-level directory. [\[Guideline\]](#)

- 13.1.1.** The source code should be placed in subdirectory with a name indicating that is what it contains, e.g., `src` [\[Guideline\]](#)

*Rationale:* It is preferable to have the source code in a subdirectory to keep the top-level one free of clutter.

- 13.1.2.** The top-level directory shall contain a file that provides basic information about the distribution. [\[Mandatory\]](#)

- 13.1.3.** It is recommended that the file be called `README`. [\[Guideline\]](#)

*Rationale:* Putting it in the top-level directory will make the file visible and give importance to it. The contents of the file should indicate how to install it, where to report issues, etc.

- 13.1.4.** Documentation for the distribution should be placed in a top-level subdirectory with a suitable name such as `doc`. [\[Guideline\]](#)

- 13.1.5.** Any scripts used to generate the top-level makefile or a configuration script should go into a suitably named subdirectory, e.g., `autoconf` for autoconf-generated scripts. [\[Guideline\]](#)

*Rationale:* These suggestions help keep the top-level directory clean allows the documentation and configuration tools to be easily found.

## 13.2 Filenames

- 13.2.0.** Filenames shall consist of only alphanumeric characters, the period, hyphen and underscore characters. No other characters are permitted. [\[Mandatory\]](#)

*Rationale:* Using characters outside this character set is forbidden because many of them need to be quoted in some way to avoid security risks after shell expansion.

- 13.2.1.** Filenames shall not contain mixed case. [\[Mandatory\]](#)

- 13.2.2.** Source code filenames shall *not* contain uppercase alphabetic characters. [\[Mandatory\]](#)

- 13.2.3.** C++ source and header filename extensions shall consist of more than one character. [\[Mandatory\]](#)

- 13.2.4.** No two files in a directory shall have the same name when lower- or upper-cased. [\[Mandatory\]](#)

*Rationale:* Since not all filesystems are case-sensitive, filenames should consist of one case or the other, e.g., `README` is permitted but `Readme` is not. The case requirement for source code files stems from the fact that some compilers treat files with a `.C` extension as a C++ file. To avoid this issue, C++ filename extensions are to consist of more than a single character, e.g., `main.cc` and `main.hh` are ok but `main.C` or `main.H` are not.

- 13.2.5.** The name of a source code file should be short, but long enough to be somewhat descriptive of the algorithms contained in it. [\[Guideline\]](#)

*Rationale:* This adds meaning to source code directory listings.