Coding Art

Release 0.0.1

charlie WONG

May 21, 2017

Contents

CHAPTER 1

C

Header Files

Self Contained Headers

- All header files should be self-contained (compile on their own) and end in `.h`.
- Non-header files that are meant for inclusion should end in `.inc` and be used sparingly.

All header files should have *header guards* and include all other header files it needs. Prefer placing the definition of **inline** functions in the same file as their declarations. The definitions of these functions must be included into every `.c` file that uses them.

The Header Guard

All header files should have `#define` guards to prevent multiple inclusion. The format of the symbol name should be `<PROJECT>_<PATH>_<FILE>_H`.

The `#define` guards should be uniqueness, based on the full path in a project's source tree. For example, the file `foo/bar/baz.h` in project **foo** should have the following guard:

```
/// @file
/// file description

#ifndef FOO_BAR_BAZ_H
#define FOO_BAR_BAZ_H
...
#endif // FOO_BAR_BAZ_H
```

Forward Declarations

Avoid using forward declarations where possible. Just `#include` the headers you need.

- Try to avoid forward declarations of entities defined in another project.
- When using a function declared in a header file, always `#include` that header.

Please see *Names and Order of Includes* for rules about when and where to `#include` a header.

Names and Order of Includes

Use standard order for readability and to avoid hidden dependencies, that is:

- C system headers

- C++ system headers
- dependency libraries headers
- your project headers

All of a project's header files should be listed as descendants of the project's source directory without use of UNIX directory shortcuts: the current directory (.) or the parent directory (..)

For example, the file `foo/bar/baz.h` in project **foo** should be included as:

```
#include "bar/baz.h"
```

Tip: Sometimes, system-specific code needs conditional includes. Such code can put conditional includes after other includes. Of course, keep your system-specific code small and localized.

Inline Functions

Define functions inline only when they are small, say, 10 lines or fewer.

You can declare functions in a way that allows the compiler to expand them inline rather than calling them through the usual function call mechanism.

- Macros for small functions are ok.
- Use inlines function instead of macros where possible and make sense.
- Inlining a function can generate more efficient code, as long as the inlined function is small.
- Feel free to inline accessors and mutators, and other short, performance-critical functions.
- Overuse of inlining can actually make programs slower.
- Depending on a function's size, it can cause the code size to increase or decrease.
- Inlining a very small accessor function will usually decrease code size while inlining a very large function can dramatically increase code size.

Tip:

- do not inline a function if it is more than 10 lines long.
- it's typically not cost effective to inline functions with loops or switch statements.
- It's important to know that functions are not always inlined even if declared as such.

Header Constants

Do not use macros to define constants in header files whenever possible.

In general `enum` are preferred to `#define` as enums are understood by the debugger. Macro constants in header files cannot be used by unit tests. However, you are allowed to define a macro that holds the same value as a non-enum constant, if the value of the constant represents the size of an array.

Be aware enums are not of a **guaranteed** size. So if you have a type that can take a known range of values and it is transported in a message you can not use an `enum` as the type. Use the correct integer size and use constants or `#define`. Casting between integers and enums is very error prone as you could cast a value not in the `enum`.

Scoping

Global Variables

Avoid using global variable where possible. Also see *Global Variable Naming Rules*.

Static Variables

When variables definied in a `.c` file do not need to be referenced outside that file, declare them as `static`. It is rarely to define `static` variables in header files. Also see *Static Variable Naming Rules*.

Locale Variables

Place a function's variables in the narrowest scope, and initialize variables in the declaration.

- Declare local variables in as local a scope as possible.
- Declare local variables as close to the first use as possible.

This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialized to. In particular, initialization should be used instead of declaration and assignment, e.g.

```
int i;
i = get_value();      // Bad  -- initialization separate from declaration.

int j = get_value(); // Good -- declaration has initialization.
```

Functions

Parameter Ordering

When defining a function, the parameters order is: **inputs**, then **outputs**.

Parameters to C functions are either input to the function, output from the function, or both. **Input** parameters are usually values or const pointers, while **output** and **input/output** parameters will be pointers to non-const.

When ordering function parameters, put all **input-only** parameters before any **output** parameters. In particular, do not add new parameters to the end of the function just because they are new, be sure placing the new **input-only** parameters before the **output** parameters.

This is not a hard-and-fast rule. Parameters that are both **input** and **output** muddy the waters, and, as always, consistency with related functions may require you to bend the rule.

Write Short Functions

Prefer small and focused functions, it is more readable and more manageable.

We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 50 lines, think about whether it can be broken up without harming the structure of the program.

Another measure of the function is the number of local variables. They shouldnâĂŹt exceed 5-10. If not, then re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know youâĂŹre brilliant, but maybe youâĂŹd like to understand what you did 2 weeks ago.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

Static Functions

When functions defined in a `.c` file, and will never be used outside of that file, then just declare them as `static`.

C Features

Variable-Length of arrays and alloca()

Variable-length arrays can cause hard to detect stack overflows. So, avoid use them where possible.

Data Struct

When declaring variables in structures, declare them organized in a manner to attempt to minimize the memory wastage because of compiler alignment issues.

- Each member variable gets its own type and line and no exception.
- Note that the use of bitfields in general is discouraged.

Major structures should be declared at the top of the file in which they are used, or in separate header files, if they are used in multiple source files. The declaration and the usage of the struct should be separated if they are used more then once.

Use Const

Use `const` whenever it makes sense.

- Easier for people to understand how variables are being used.
- Allows the compiler to do better type checking, and generate better code.
- Helps people convince the program correctness because they know the functions they call are limited in how they can modify the variables.
- Helps people know what functions are safe to use without locks in multi-threaded programs.

Use `const` pointer whenever it makes sense..

Some people favor the form `int const *ptr` to `const int *ptr`.

They argue that this is more readable because it's more consistent: it keeps the rule that `const` always follows the object it's describing. However, this consistency argument doesn't apply in codebases with few deeply-nested pointer expressions since most const expressions have only one `const`. In such cases, there's no consistency to maintain. Putting the `const` first is arguably more readable, since it follows English in putting the **adjective** (`const`) before the **noun** (`int`).

While we encourage putting `const` first, do not require it, just be consistent with the code around you!

Integer Types

The built-in integer types:

> `char`, `int`, `uint8_t`, `int8_t`, `uint16_t`, `int16_t`, `uint32_t`, `int32_t`, `uint64_t`, `int64_t`, `uintmax_t`, `intmax_t`, `size_t`, `ssize_t`, `uintptr_t`, `intptr_t`, `ptrdiff_t`

Use `int` for error codes, local and trivial variables only.

BE care when converting integer types. Integer conversions and promotions can cause non-intuitive behavior.

Note that the signedness of `char` is implementation defined.

There are no convenient `printf` format placeholders for fixed width types. Cast to `uintmax_t` or `intmax_t` if you have to format fixed width integers.

Type	unsigned	signed
char	%hhu	%hhd
int	n/a	%d
(u)intmax_t	%ju	%jd
(s)size_t	%zu	%zd
ptrdiff_t	%tu	%td

Tip:

- If needs different size of integers, use a precise-width integer type from `<stdint.h>`.
- If a variable represents a value that could ever be greater than or equal to 2^31 (2GiB), use a 64-bit type, such as `int64_t`.
- Keep in mind that even if your value won't ever be too large for an int, it may be used in intermediate calculations which may require a larger type.

Use Boolean

Use boolean type to represent boolean values when it is possible.

```
int  is_good = 1;      // Bad  - should have type bool.
bool is_good = true;   // Good - just use boolean type where possible.
```

Variable Declarations

Declare only one variable per line, and each line have only one sentence.

```
int i, j = 1;      // never do this
int k=0; func();   // never do this also

int i = 0;         // this will more clear
int j = 1;         // no bug easy to hide
int k = 0;
func();
```

0 and NULL & NUL

- Use `0.0` for real
- Use `0` for integer
- Use `NULL` for pointer

- Use `'\0'` or `NUL` for char

Usage of sizeof

Prefer `sizeof(varname)` to `sizeof(type)`.

Use `sizeof(varname)` when you take the size of a particular variable. `sizeof(varname)` will update appropriately if someone changes the variable type either now or later.

You may use `sizeof(type)` for the code unrelated to any particular variable.

Usage of goto

Just do not use `goto` when it is absolutely necessary.

The `goto` statement comes in handy when a function exits from multiple locations and some common work such as cleanup has to be done. If there is no cleanup needed then just return directly.

The abusively use of `goto` make code hard to read and management, so just use it as rare as possible. If for some reason, you must use `goto`, then choose label names which say what the `goto` does or why the `goto` exists.

The rationale for using `gotos` is:

- nesting is reduced.
- errors by not updating individual exit points when making modifications are prevented.
- saves the compiler work to optimize redundant code away.

A common type of bug to be aware of is **one err** bug which look like this:

```
err:
    kfree(foo->bar);
    kfree(foo);
    return ret;
```

The bug in this code is that on some exit paths `foo` is `NULL`. Normally the fix for this bug is to split it up into two error labels `err_free_bar` and `err_free_foo`, e.g.

```
err_free_bar:
    kfree(foo->bar);
err_free_foo:
    kfree(foo);
    return ret;
```

Usage of Macros

Macros with multiple statements should be enclosed in a `do { ... }while(0)`, so that a trailing semicolon works, e.g.

```
// make properly align of \'s to increase the readability
#define macrofun(a, b, c)    \
    do                       \
    {                        \
        if(a == 5)           \
        {                    \
            do_this(b, c);   \
        }                    \
    }while(0)
```

- Avoid using macros if they affect control flow, e.g.

```
// do not do it like this
#define FOO(x)                  \
    do                          \
    {                           \
        if(blah(x) < 0)         \
        {                       \
            return ERROR_CODE;  \
        }                       \
    }while(0)
```

- Avoid using macros if they depend on having a local variable with a magic name, e.g.

```
// what the hell of the 'index' and 'val'
#define FOO(val) bar(index, val)
```

- Make the expression precedence very very clear by using properly parentheses.
- Macros should be used with caution because of the potential for error when invoked with an expression that has side effects.
- When putting expressions in macros always wrap the expression in parenthesis to avoid potential communitive operation abiguity.

```
#define max(x, y)  ((x>y)?(x):(y))
...
max(f(x), z++);
```

Conditional Compilation

Wherever possible, donâĂŹt use preprocessor conditionals (`#if`, `#ifdef`, etc.) in `.c` files, and doing so makes code harder to read and logic harder to follow. Instead, use such conditionals in a header file defining functions for use in those `.c` files, providing no-operation stub versions in the `#else` case, and then call those functions unconditionally from `.c` files. The compiler will avoid generating any code for the stub calls, producing identical results, but the logic will remain easy to follow.

If you have a function or variable which may potentially go unused in a particular configuration, and the compiler would warn about its definition going unused, so just mark the definition with `__attribute__((unused))` (see See GCC Attribute Syntax) rather than wrapping it in a preprocessor conditional. However, if a function or variable always goes unused, then just delete it.

At the end of any non-trivial `#if` or `#ifdef` block (more than a few lines), place a comment after the `#endif` on the same line, noting the conditional expression used. For instance:

```
#ifdef CONFIG_SOMETHING

doing_some_thing

#endif // CONFIG_SOMETHING
```

Also do NOT not put `#ifdef` in an expressions for readability.

Mixing C and C++

When calling a C function from C++, the function name will be mangled unless you turn it off. Name mangling is turned off with the `extern "C"` syntax.

- If you want to create a C function in C++ you must wrap it with the `extern "C"` syntax.
- If you want to call a C function in a C library from C++ you must wrap in the above syntax.

```
//Creating a C function in C++
extern "C" void
a_c_function_in_cplusplus(int a)
{
    do_some_thing();
}

// Calling C functions from C++
extern "C" int strncpy(...);
extern "C" int my_great_function();
extern "C"
{
    int strncpy(...);
    int my_great_function();
};
```

If you have code that must compile in a C and C++ environment then you must use the `__cplusplus` preprocessor directive, e.g.

```
#ifdef __cplusplus
    extern "C" some_function();
#else
    extern some_function();
#endif
```

Naming

The most important consistency rules are those that govern naming. The style of a name immediately informs us what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc., without requiring us to search for the declaration of that entity. The pattern-matching engine in our brains relies a great deal on these naming rules.

A name is the result of a long deep thought process about the ecology it lives in. Only a programmer who understands the system as a whole can create a name that **fits** with the system. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected.

Naming rules are pretty arbitrary, but we feel that consistency is more important than individual preferences in this area, so regardless of whether you find them sensible or not, the rules are the rules.

General Naming Rules

- Use intention-revealing names, names should reveal real intent, avoid disinformation.
- Make names should have meaningful distinctions and try best make them pronounceable.
- Make it clear, easy to read, write and understand, within reason.
- Names should be descriptive, avoid abbreviation.
- Give more context information in the names.

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader.

Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do NOT **abbreviate** by deleting letters within a word. Note that certain universally-known abbreviations are OK, such as `i` for an iteration variable.

```
int price_count_reader;    ///< Good - No abbreviation.
int num_errors;            ///< Good - "num" is a widespread convention.
```

```
int num_dns_connections;    ///< Good - Most people know what "DNS" stands for.

int n;                      ///< Bad  - Meaningless.
int nerr;                   ///< Bad  - Ambiguous abbreviation.
int n_comp_conns;           ///< Bad  - Ambiguous abbreviation.
int wgc_connections;        ///< Bad  - Only your group knows what this stands for.
int pc_reader;              ///< Bad  - Lots of things can be abbreviated "pc".
int cstmr_id;               ///< Bad  - Deletes internal letters.
```

File Naming Rules

Filenames should be all lowercase with hyphens between words, if necessary can have digital numbers. Source files should end in `.c`, and header files should end in `.h`. Examples of acceptable file names are:

- my-useful-file.c
- my-useful-file.h
- Do not reuse a standard header file names, e.g. **locale.h**, **time.h**, etc.
- In general, make your filenames very specific, with more context information. For example, use **http-server-logs.h** rather than **logs.h**.

Type Naming Rules

Typedef-ed types should be all lowercase with underscores between words, and have one of suffix:

- suffix `_st` for `typedef struct`
- suffix `_et` for `typedef enum`
- suffix `_ut` for `typedef union`
- suffix `_bt` for basic type aliases, e.g. `typedef int buffer_id_bt;`
- suffix `_ft` for function type, e.g. `typedef int (*func_ft)(int cnt);`

Non-Typedef-ed `struct`/`enum`/`union` for forward necessary declaration, with one of suffix as following:

- suffix `_ST` for `struct`
- suffix `_ET` for `enum`
- suffix `_UT` for `union`

Use typedefs of non-pointer types only, , and all use typedef version, for convenience.

Common Variable Naming Rules

All variable names consist of lowercase and underscores, if necessary can have digital numbers.

For example:

```
string table_name;  ///< OK  - uses underscore.
string tablename;   ///< OK  - all lowercase.

string tableName;   ///< Bad - mixed case.
```

Tip:

- It is a good idea to make and use searchable names.

Struct Member Naming Rules

Members of `struct` are named like *common variables* with prefix `m_`.

Enum Member Naming Rules

Members of `enum` are named like *common variables* with prefix `k_`.

No comma on the last element of `enum`, e.g.

```
enum my_enum_ET
{
    k_me_one,
    k_me_two,
    ...
    // no comma on the last element
    k_me_last
};
```

Tip: It maybe a good idea to hava format like, `k_<id>_`, where `id` is a short name derived from that enumeration.

Union Member Naming Rules

Members of union are named like *common variables* with prefix `m_`.

Global Variable Naming Rules

Global variable name just like *common variables*, but with prefix `g_`.

Static Variable Naming Rules

Static variable name just like *common variables*, but with prefix `s_`.

Local Variable Naming Rules

Local variable just following *Common Variable Naming Rules*.

Local variable names should be short, and to the point. If you have some random integer loop counter, it should probably be called `i`. It is clear enough and there is no chance of it being mis-understood. Similarly, `tmp` can be just about any type of variable that is used to hold a temporary value.

Constant Variable Naming Rules

All constant variables, and whose value is fixed for the duration of the program, following *Common Variable Naming Rules*, but with a leading `k`. Also see *Enum Member Naming Rules*.

```
const int kweeks_days = 7;
const int kday_hours  = 24;
```

Function Naming Rules

Function names consist of lowercase and underscores, if necessary can have digital numbers.

Usually every function performs an action, so the name should make clear what it does, for example: `check_for_errors()` is better than `error_check()`, `dump_data_to_file()` instead of `data_file()`. This will also make functions and data objects more distinguishable.

Structs are often nouns. By making function names verbs and following other naming conventions programs can be read more naturally.

Suffixes & Prefixes are sometimes useful:

- It maybe a good idea to have a perfix for a serial or module of functions.
- `_max` - to mean the maximum value something can have.
- `_cnt` - the current count of a running count variable.
- `_ptr` - the pointer variable.
- `is_` - to ask a question about something.
- `get_` - get a value.
- `set_` - set a value.

Macro Naming Rules

Macro names consist of uppercase and underscores, if necessary can have digital numbers.

- If macros are resembling functions, then name them in lower case is better.
- If a macros can be empty, then always use capitalized letters, e.g. `DEBUG_MSG(msg)`.

```
/// header file guard macro
#define <PROJECT>_<PATH>_<FILE>_H

/// awesome macro defination
#define AWESOME_MACRO_DEFINATION

/// constant number value
#define PI (3.1415926)

/// constant string value
#define CONFIG_FILE_NAME "config"

/// function like macro
#ifdef SHOW_DEBUG_MESSAGE
#   define DEBUG_MSG(msg) printf("%s\n", msg);
#else
#   define DEBUG_MSG(msg)
#endif
```

Note:

- General speaking, if not necessary, macros should not be used.
- Properly use of inline functions instead of macro functions make sense.

Comments

Though a pain to write, comments are absolutely vital to keeping our code readable. The following rules describe what you should comment and where.

If we make a step further, just do little work while writing comments, then we can use the doc-tools out there to automatically generating perfect documentation. Here we use the well known documentation tool Doxygen.

When writing comments, write for your audience: the next contributor who will need to understand your code.

Be generous - the next one may be you!

Tip:

- While comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that you must then explain through comments.
- do not abuse comments. Comments are good, but there is also a danger of over-comment. NEVER try to explain **HOW** your code works in a comment: itâĂŹs much better to write the code so that the working is obvious, and itâĂŹs a waste of time to explain badly written code.

Comment Style

Prefer using the `//` style syntax only, much more easier to type.

```
// This is a comment spanning
// multiple lines
func();
```

Tip:

- Use either the // or /* */ syntax, as long as you are consistent.
- `//` style syntax is not for extracting comments, use `///` syntax if you want to do that.

File Comments

Every file should have a comment at the top describing its contents. For example:

```
/// @file
/// A brief description of this file.
///
/// A longer description of this file.
/// Be very generous here.
```

Generally speaking:

- Comments in `.h` file will describe the variables and functions that are declared in the file with an overview of what they are for and how they are used.
- Comments in `.c` file should contain more information about implementation details or discussions of tricky algorithms. If you feel the implementation details or a discussion of the algorithms would be useful for someone reading the `.h`, feel free to put it there instead, but mention in the `.c` that the documentation is in the `.h` file.

> **Warning:** Do not duplicate comments in both the `.h` and the `.c`. Duplicated comments diverge.

Structured Data Comments

Every `struct`, `enum` and `union` definition should have accompanying comments that describes what it is for and how it should be used.

If the field comments are short, you can put them next to the field, for example:

```
/// Structure used for growing arrays.
///
/// This is used to store information that only grows, and deleted all at once,
/// and needs to be accessed by index. Also see @ref ga_clear() and @ref ga_grow().
typedef struct growarray
{
    int   m_ga_size;       ///< current number of items used
    int   m_ga_maxsize;    ///< maximum number of items possible
    int   m_ga_itemsize;   ///< sizeof(item)
    int   m_ga_growsize;   ///< number of items to grow each time
    void *m_ga_data;       ///< pointer to the first item
}garray_st;
```

If the field comments are long, you can put them previous to the field, for example:

```
/// ...
typedef struct growarray
{
    /// current number of items used
    int   m_ga_size;
    /// maximum number of items possible
    int   m_ga_maxsize;
    /// sizeof(item), item size in bytes
    int   m_ga_itemsize;
    /// number of items to grow each time
    int   m_ga_growsize;
    /// pointer to the first item
    void *m_ga_data;
}garray_st;
```

Function Declarations Comments

Comments at the declaration of a function describe the **usage** of the function. Every function declaration should have comments immediately preceding it that describe what the function does and how to use it. In general, these comments do not describe how the function performs its task which should be left to comments in the function definition.

Types of things to mention in comments at the function declaration:

- Whether the function allocates memory that the caller must free.
- Whether any of the arguments can be a null pointer.
- Whether there are any performance implications of how a function is used.
- Whether the function is re-entrant.
- What are its synchronization assumptions.

```
/// Brief description of the function.
///
/// Detailed description.
/// May span multiple paragraphs.
///
/// @param[in] arg1 Description of arg1
/// @param[in] arg2 Description of arg2. May span
///                 multiple lines.
```

```
///
/// @return Description of the return value.
iterator_st *get_iterator(void *arg1, void *arg2);
```

Function Definition Comments

Comments at the definition of a function describe **operation** of the function. If there is anything tricky about how a function does its job, the function definition should have an explanatory comment. For example, in the definition comment you might describe any coding tricks you use, give an overview of the steps you go through, or explain why you chose to implement the function in the way you did rather than using a viable alternative. For instance, you might mention why it must acquire a lock for the first half of the function but why it is not needed for the second half.

```
// Note that do not use Doxygen comments here. They are not for Doxygen.
iterator_st *get_iterator(void *arg1, void *arg2)
{
...
}
```

Note: Do not just repeat the comments given with the function declaration, in the `.h` file or wherever. It's okay to recapitulate briefly what the function does, but the focus of the comments should be on how it does it.

Variable Comments

In general the actual name of the variable should be descriptive enough to give a good idea of what the variable is used for. In certain cases, more comments are required.

```
/// The total number of tests cases that we run through in this regression test.
const int g_test_cases_num = 6;
```

- All global variables should have comments describing what they are and what they are used for.

Implementation Comments

In your implementation, you should have comments in tricky, non-obvious, interesting, or important parts of your code.

Explanatory Comments: tricky or complicated code blocks should have comments before them.

```
// Divide result by two, taking into account that x contains the carry from the add.
for(int i = 0; i < result->m_size; i++)
{
    x = (x << 8) + (*result)[i];
    (*result)[i] = x >> 1;
    x &= 1;
}
```

Line Comments: lines that are non-obvious should get a comment at the end of the line. These end-of-line comments should be separated from the code using spaces.

```
// If we have enough memory, mmap the data portion too.
mmap_budget = max<int64>(0, mmap_budget - index->m_length);
if(mmap_budget >= data_size && !map_data(mmap_chunk_bytes, mlock))
{
    return;  // Error already logged.
}
```

Line Up Comments: if you have several comments on subsequent lines, it can often be more readable to line them up:

```
do_something();                         // Comment here so the comments line up.
do_something_else_that_is_longer();     // Comment here so there are two spaces between
                                        // the code and the comment.
{
    do_something_else();                // Comment here so the comments line up.
}
```

No Magic Arguments: when you pass in a null pointer, boolean, or literal integer values to functions, you should consider adding a comment about what they are, or make your code self-documenting by using constants. For example, compare:

```
bool success = calculate_something(interesting_value,
                                   10,      // What is this ?
                                   false,   // What is this ?
                                   NULL);   // What is this ?
```

versus:

```
bool success = calculate_something(interesting_value,
                                   10,      // Default base value.
                                   false,   // Not the first time we're calling this.
                                   NULL);   // No callback.
```

Or alternatively, constants or self-describing variables:

```
// line them up make more readable, both definition and comments
const int     default_base_value = 10;      // Default base value.
const bool    first_time_calling = false;   // Not the first time calling this.
callback_ft   null_callback = NULL;         // No callback

bool success = calculate_something(interesting_value,
                                   default_base_value,
                                   first_time_calling,
                                   null_callback);
```

Tip:

- Never describe the code itself, just assume the reader knows C better than you.
- Never abuse comment, do not state the obvious.
- Provide higher level comments that describe why the code does what it does.

Punctuation, Spelling and Grammar

Pay attention to punctuation, spelling, and grammar. It is easier to read well-written comments than badly written ones.

Comments should be as readable as narrative text, with proper capitalization and punctuation. In many cases, complete sentences are more readable than sentence fragments. Shorter comments, such as comments at the end of a line of code, can sometimes be less formal, but you should be consistent with your style.

Although it can be frustrating to have a code reviewer point out that you are using a comma when you should be using a semicolon, it is very important that source code maintain a high level of clarity and readability. Proper punctuation, spelling, and grammar help with that goal.

TODO Comments

Use TODO comment for code that is temporary, short-term solution, or good-enough but not perfect.

TODO comment should include the string **TODO** or **todo**, followed by the name, e-mail address, bug ID, or other identifier(person or issue), which can provide best context about the problem referenced by the TODO Comment. The main purpose is to have a consistent TODO comment format that can be searched to find out how to get more details upon request. A TODO comment is not a commitment that the person referenced will fix the problem. Thus when you create a TODO comment with a name, it is almost always your name that is given.

```
/// @todo (kl@gmail.com): Use a "*" here for concatenation operator.
/// @todo (Zeke): change this to use relations.
/// @todo (bug 12345): remove the "Last visitors" feature.

//  TODO (kl@gmail.com): Use a "*" here for concatenation operator.
//  TODO (Zeke): change this to use relations.
//  TODO (bug 12345): remove the "Last visitors" feature.
```

Deprecation Comments

Use Deprecation Comment for the interface API that is deprecated.

You can mark an interface as deprecated by writing a comment containing the word **DEPRECATED** or **deprecated**, followed by your name, e-mail address, or other identifier in parentheses. The comment goes either before the declaration of the interface or on the same line as the declaration.

A deprecation comment must include simple, clear directions for people to fix their callsites. In C, you can implement a deprecated function as an inline function that calls the new interface point.

Marking an interface point deprecated will not magically cause any callsites to change. If you want people to actually stop using the deprecated facility, you will have to fix the callsites yourself or recruit a crew to help you.

New code should not contain calls to deprecated interface points. Use the new interface point instead. If you cannot understand the directions, find the person who created the deprecation and ask them for help using the new interface point.

Formatting

Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

Line Length

- Each line of text in your code should be at most 100 characters long.
- Use `\n` as the new line sign only, no `\r\n` or `\r`.

Note:

- Maybe you are not agree with 100 as maxlength of lines, exactly is 99 visiable characters with one invisiable character(new line sign), and prefer to 80, 120 or others. Thus, regardless of whether you find them sensible or not, the rules are the rules.
- If you are writing for print using A4, changing this back to 80 maybe more reasonable, because the max character length of A4 is 80.

- Comment lines can be longer than 100 characters if it is not feasible to split them without harming readability. e.g. if a line contains an example command or a literal URL longer than 100 characters.
- A raw-string literal may have content that exceeds 100 characters.
- An `#include` statement with a long path may exceed 100 columns.

Indentation

Tabs are 4 characters, and thus indentations are also 4 characters, and use `spaces` only.

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

If having 4-character indentations makes the code move too far to the right, and makes it hard to read on the screen, then it maybe a warning that your logical have problems and you need to find an easy way to do that, and fix your problem.

Tip:

- DonâĂŹt put multiple statements on a single line unless you have something to hide.
- DonâĂŹt put multiple assignments on a single line either.
- Avoid complicated expressions.

Spaces VS. Tabs

Use only spaces, and indent 4 spaces at a time. Do not use **tabs** in your code.

- You should set your editor to emit spaces when you hit the tab key.
- Put it this way will make the code format always correct with all text editors.

Warning: Use tabs when it is necessary, such as the Makefile rules.

Non-ASCII Characters

Non-ASCII characters should be rare, and always use UTF-8 formatting.

You shouldn't hard-code user-facing text in source, even English, so use of non-ASCII characters should be rare. However, in certain cases it is appropriate to include such words in your code. For example, if your code parses data files from foreign sources, it may be appropriate to hard-code the non-ASCII string(s) used in those data files as delimiters. More commonly, unittest code (which does not need to be localized) might contain non-ASCII strings. In such cases, you should use UTF-8, since that is an encoding understood by most tools able to handle more than just ASCII.

Hex encoding is also OK, and encouraged where it enhances readability, for example, `\uFEFF`, is the Unicode **zero-width no-break space** character, which would be invisible if included in the source as straight UTF-8.

Breaking Long Lines and Strings

Coding style is all about readability and maintainability using commonly available tools.

Statements longer than 100 columns will be broken into sensible chunks, unless exceeding 100 columns significantly increases readability and does not hide information. Descendants are always substantially shorter than the

parent and are placed substantially to the right. However, never break user-visible strings messages, because that breaks the ability to **grep** for them.

The Usage of Braces

Put the opening and closing brace on the line just by itself, for all statement blocks, thusly:

```
if(is_true)
{
    do_it_like_this();
    // decrease the code density, make the start/end more clear
}
else if(is_good)
{
    do_some_thing();
}
else
{
    do_other_thing();
}

struct this_is_good
{
    bool m_good;
    ...
}
```

Note that the closing brace is empty on a line of its own, the only exception is it followed by a continuation, that is a `do-while` statement, e.g.

```
do
{
    do_it_like_this();
}while(is_true);
```

Prefer curly brace where a single statement is enough, make it clear enough, e.g:

```
if(condition)
{
    action();
}

if(condition)
{
    do_something();
}
else
{
    do_another();
}
```

Adding short comment to closing braces properly may be helpful when you are reading code chunks, because you don't have to find the begin brace to know what is going on especially for *big* and *long* code blocks.

Tip:

- clearness and readability is much more important.
- do not worried about saving lines.

The Usage of Spaces

- NO spaces after the keywords, the notable exceptions of C and the function names.
- NO spaces after-the-open and before-the-close parentheses.
- NO space around the `.` and `->` structure member operators.

```
// Keywords of C
if, switch, case, for, do, while

// Notable exceptions of C
sizeof, typeof, alignof, __attribute__
```

```
// do not need to emphasis the keywords, it is clear enough
while (condition)
{
    do_something();
}

// do not need to emphasis the condition, it is clear enough
if( condition )
{
    do_something();
}

s = sizeof( struct file ); // This is not good.
s = sizeof( struct file ); // This is good enough.
```

- Use one space around (on each side of) most binary and ternary operators, such as any of these:

```
=  +  -  <  >  *  /  %  |  &  ^  <=  >=  ==  !=  ?  :
```

- NO space after unary operators, such as any of these:

```
&  *  +  -  ~  !  sizeof  typeof  alignof  __attribute__  defined
```

- NO space before the postfix increment and decrement unary operators:

```
++  --
```

- NO space after the prefix increment and decrement unary operators:

```
++  --
```

Note: Although, for notable exceptions, the parentheses are not required in the language, for example, `sizeof info;` is the same as `sizeof(info);` after `struct fileinfo info;` is declared, it will make things simple by using parentheses all the time.

The Usage of Stars

When declaring pointer variable or a function that returns a pointer type, the preferred use of `*` is adjacent to the variable name or function name and not adjacent to the type name, e.g:

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

Function Declarations and Definitions

- Return type on the same line as function name.

- Parameters on the same line if they fit.
- Wrap parameter lists which do not fit on a single line.

Function on the same line, for example:

```
return_type function_name(type arg_name_1, type arg_name_2)
{
    do_something();
    ...
}
```

Function on more then one line, too much text to fit on one line, for example:

```
return_type function_name_1(type arg_name_1, type arg_name_2, type arg_name_3,
                            type arg_name_4)
{
    do_something();
    ...
}

return_type function_name_2(type arg_name_1, type arg_name_2, type arg_name_3,
                            type arg_name_4, type arg_name_5, type arg_name_6)
{
    do_something();
    ...
}
```

- Choose good parameter names.
- The open parenthesis is always on the same line as the function name.
- There is never a space between the function name and the open parenthesis.
- There is never a space between the open parentheses and the first parameters.
- The open curly brace is always on the next line by itself.
- The close curly brace is always on the last line by itself.
- All parameters should be named, with identical name in declaration and implementation.
- All parameters should be aligned if possible.
- Default indentation is 4 spaces.
- Wrapped parameters should indent to the function's first arguments.

Tip: Maybe it is time to rewrite the function interface by group the arguments into a struct if it has too much text to fit on one line.

Function Calls

Write the call all on a single line if it fits, function calls have the following format:

```
bool retval = do_something(arg_1, arg_2, arg_3);
```

If the arguments do not fit on one line, they should be broken up onto multiple lines, with each subsequent line aligned with the first argument. Do not add spaces after the open paren or before the close paren:

```
bool retval = do_something(a_very_very_very_very_long_arg_1,
                           arg_2, arg_3);
```

If the function has many arguments, consider having one per line if this makes the code more readable:

```
bool retval = do_something(arg_1,
                           arg_2,
                           arg_3,
                           arg_4);
```

If the function has many arguments, consider having minimum number of lines by breaking up onto multiple lines, with each subsequent line aligned with the functions's first argument:

```
bool retval = do_something(arg_1, arg_2, arg_3, arg_4
                           arg_5, arg_6, arg_7, arg_8);
```

Arguments may optionally all be placed on subsequent lines, with one line per argument:

```
if(...)
{
    do_something(arg_1,
                 arg_2,
                 arg_3,
                 arg_4);
}
```

Braced Initializer List

Format a braced list exactly like you would format a function call in its place.

If the braced list follows a name (e.g. a type or variable name), format as if the `{ }` were the parentheses of a function call with that name. If there is no name, assume a zero-length name.

```
struct my_struct_ST m =
{
    superlongvariablename_1,
    superlongvariablename_2,
    { ages , interior, list },
    {
        interiorwrappinglist_1,
        interiorwrappinglist_2,
    }
};
```

Conditionals

- Prefer no spaces inside parentheses.
- The `if`, `else` and `if else` keywords belong on separate lines by itself, no curly.
- Always use curly braces, even if the body is only one sentence.
- Make 4 space indent, make sure no use tabs.
- Make sure there is no space between `if`/`else`/`if else` keywords and the open parentheses.

```
// Good - no spaces inside parentheses
// Good - no spaces between if and the open parentheses
// Good - if just on the line by itself
if(condition)
{       // Good - open curly on the next line by itself
    ...  // Good - 4 space indent
}       // Good - close curly on the last line by itself
else if(...)
{
    ...
}
```

```
else
{
    ...
}

if( condition ) // Bad - have two spaces inside parentheses
{
  do_some();    // Bad - not 4 space indent
  ...
}
else if(...) {  // Bad - open curly and else-if not on the line just by itself
    ...
}
else {          // Bad - else/open curly not on the line just by itself
    ...
}
```

Even if the body is only one sentence, the curly can still not be omitted. Never use a single sentence or empty curly as the body, so the single semicolon.

```
if(x == foo) { return foo(); } // Good - this will be fine.
if(x == foo)
{
    return foo();              // Good - clear enough.
}

if(x == bar) bar();            // Bad  - this is not good, easy misreading
do_another_thing();

if(x == bar) return bar();     // Bad  - no curly.
if(x == bar) {}                // Bad  - do you realy need this?
```

Loops and Switch Statements

- Empty loop bodies should only use an `continue` inside curly.
- Never use a single sentence or empty curly as the body, so the single semicolon.

```
while(condition) { continue; }          // Good - continue indicates no logic.
while(condition)
{
    continue;                           // Good - clear enough.
}

while(condition) {}                     // Bad  - is this part finished?
for(int i = 0; i < some_number; i++) {} // Bad  - why not do it in the body?
while(condition);                       // Bad  - looks like part of do/while loop.
```

- Single-statement loops should always have curly braces.

```
for(int i = 0; i < some_number; ++i)
{
    printf("I take it back\n");  // Good - 4 space indent
}

while(condition)
{
    do_something();              // Good - 4 space indent
}

for(int i = 0; i < some_number; ++i)
    printf("I love you\n");      // Bad   - no braces
```

```
for(int i = 0; i < some_number; ++i)
{
  printf("I take it back\n");    // Bad  - not 4 space indent
}
```

- `case` blocks in `switch` statements should always have curly braces.
- align the subordinate `case` labels in the same column with `switch`.
- `switch` statements should always have a `default` case, no exception.
- No space before the colon of `case`.
- If the `default` case should never execute, simply `assert`.

```
switch(var)
{           // open curly braces must on the next line by itself
case 0:     // each case must 4 space indent
{
    ...
    break;  // 4 space indent
}
case 1:     // no space before the colon
{
    ...
    break;
}
default:
{
    assert(false);
}
}

switch(var)
{
// for readability, this is also good
case 0:  do_some_thing_short(); break;
case 1:  another_thing_short(); break;
default: assert(false);
}
```

Tip: The space around the operator in loop condition is optional and feel free to insert extra parentheses judiciously for readability.

Pointer Expressions

- No spaces around period or arrow.
- Pointer operators do not have trailing spaces.
- Pointer operators have no space after the `*` or `&`.

Examples of correctly-formatted pointer:

```
int  x = *p;
int *z = &x;
int  z = g.y;
int  h = r->y;
```

- When declaring a pointer variable or argument, place the asterisk adjacent to the variable name.

```
char *c;   // Good - variable name just following *, no spaces between them.

char * c;  // Bad  - spaces on both sides of *.
char* c;   // Bad  - space between * and the variable name.
```

- It is not allowed to declare multiple variables in the same declaration.

```
int x, y;  // Bad  - no multiple variables on a declaration.
int a, *b; // Bad  - such declarations are easily misread.

int  x = 2;    // Good - only one variable on a declaration.
int  y = 0;    // Good - easily initiallize it, no misreading.
int  a = 1;
int *b = NULL; // Good - such declaration clear enough.
```

- It is a bad idea to have multiple sentences on the same line.

```
// Bad  - why do you want to do like this?
int x=foo(); char c = get_char();
int a=1; char *str="good";

// Good - why do you make it clear?
int  x = foo();
char c = get_char();
int  a = 1;
char *str = "good";
```

Boolean Expressions

When a boolean expression that is longer than the standard *line length*, break it up by:

- keep operators at the end of the line, and align them for readability and emphasis.
- make all items indent to the first item of the boolean expression.

```
// use minimal lines
if(this_one_thing > this_other_thing && a_third_thing == a_fourth_thing &&
   yet_another_thing && the_last_thing)
{
    // 'yet_another_thing' align to 'this_one_thing'
    ...
}

// each on a single line, make the operator indented
if(this_one_thing > this_other_thing &&
   a_third_thing == a_fourth_thing   &&
   yet_another_thing                 &&
   the_last_thing)
{
    // all items align to 'this_one_thing'
    ...
}
```

Note:

- Be consistent in how breaking up the lines with the codes around.
- Feel free to insert extra parentheses judiciously because they can be very helpful in increasing readability when used appropriately.
- Always use the punctuation operators, such as `&&` and `~`, rather than the word operators, such as `and` and `compl`.

Return Values

- Do not needlessly surround the return expression with parentheses.
- Use parentheses in return **expr** only where you would use them in `x = expr` like format.

```
return result;                // Good - No parentheses in the simple case.
return (ret == true);         // Good - return boolean value.
return (sec : opt_1 ? opt_2); // Good - select one as the return value.

// Good - Parentheses OK to make a complex expression more readable.
return (some_long_condition && another_condition);
return (some_long_condition &&
        another_condition   &&
        yes_the_last_one);

return (value);     // Bad - You would never write 'var = (value);', would you ?
return(result);     // Bad - return is not a function!
```

Tip:

- Feel free to insert extra parentheses judiciously because they can be very helpful in increasing readability when used appropriately.

Preprocessor Directives

- The hash mark that starts a preprocessor directive should always be at the beginning of the line.
- Nested directives should make properly align after the hash mark for each level of indentation.
- If preprocessor directives are within the body of indented code, make judiciously indent to increase the readability.

```
if(lopsided_score)
{
#if DISASTER_PENDING
    drop_every_thing();
    // judiciously indent, more readable
    #if NOTIFY
    notify_client();
    #endif
#endif
    BackToNormal();
}

#ifdef DEBUG_LOG_ENABLE
#   define DEBUG_MSG(msg) printf("%s\n", (msg)); // add 3 spaces before 'define'
#else
#   define DEBUG_MSG(msg)                        // make it more readable
#endif
```

General Horizontal Whitespace

- Use of horizontal whitespace depends on location.
- Never put trailing whitespace at the end of a line.

```
int i = 0;         // Semicolons usually have no space before them.
int x[] = { 0 };   // Spaces inside braces for braced-init-list on both sides.
```

Some editors with **smart** indentation will insert whitespace at the beginning of new lines as appropriate, so you can start typing the next line of code right away. However, if some such editors do not remove the whitespace when you end up not putting a line of code there, such as if you leave a blank line. As a result, you end up with lines containing trailing whitespace.

> **Warning:** Adding trailing whitespace can cause extra work for others editing the same file when they merge, as they can removing existing trailing whitespace, they are invisiable, are't they.
> Thus, do NOT introduce trailing whitespace. Remove it if you're already changing that line, or do it in a separate clean-up operation(preferably when no-one else is working on the file.

Blocks Horizontal Whitespace

```
// no space after the keyword in conditions and loops
if(b)
{
    ...
    do_some_thing(); // 4 space indent
}

// usually no space inside parentheses
// no space after the keywords: while
while(test) { continue; }

// no space after the keywords: for
// for loops always have a space after the semicolon
// for loops usually no space before the semicolon
for(int i = 0; i < 5; ++i)
{
    // one space before the semicolon
    for( ; ; )
    {
        ...
        if(condition) break; // 4 space indent
    }
}

// no space after the keywords: switch
switch(i)
{
case 1:  // No space before colon in a switch case.
{ ... }
case 2:
{ ... }
default: // Always have default
{ ... }
}

// the same goes for union and enum
struct my_struct_ST
{   // open curly brace on the next line by itself
    // 4 space indent
    const char *m_name;  ///< name of people, max len is 100
    const char *m_addr;  ///< home address, max len is 512
    // make properly align of members
    // make properly align of members comments if have
    bool  m_boy;         ///< boy: @b true; girl: @b false
    int   m_age;         ///< age, [1, 150]
};// no space between close curly brace and semicolon

// the same goes for union and enum
```

```
typedef struct
{
    const char *m_name;  ///< name of people, max len is 100
    const char *m_addr;  ///< home address, max len is 512
    bool  m_boy;         ///< boy: @b true; girl: @b false
    int   m_age;         ///< age, [1, 150]
} my_struct_st;
// no space between the name and semicolon
// one space between close curly brace and the name
```

Operators Horizontal Whitespace

```
x = 0;              // assignment operators always have spaces around them.
v = w * x + y / z; // binary operators usually have spaces around them.
v = w*x + y/z;      // it's OK to remove spaces around factors, if still clear enough.
v = w * (x + z);    // parentheses should have no internal padding.

// no spaces separating unary operators and their arguments.
x = -5;
++x;
if(x && !y)
{
    ...
}
```

Tip:

- Feel free to insert extra parentheses judiciously because they can be very helpful in increasing readability when used appropriately.

Variables Horizontal Whitespace

```
int long_variable = 0;  // NEVER align assignments like this.
int i             = 1;

int i = 1;              // this will be clear and good enough.
int a_var = 0;
int an_var = 1;
int yes_anox = 5;
int long_variable = 0;

struct my_struct_ST
{
    const char *m_name;
    const char *m_addr;  // make properly align of members
    bool  m_boy;         // make properly align of members
    int   m_age;
} my_variable[] =        // one space between close curly brace and variable
{   // open curly brace on the next line by itself
    // make properly align, increasing the readability
    // make sure no space before the comma
    // 4 space indent
    { "Mia",       "Address",        true,  8  },
    { "Elizabeth", "AnotherAddress", false, 10 },
};
```

Macros Horizontal Whitespace

```
// Align \'s in macro definitions like this, increasing readability
#define __KHASH_TYPE(name, khkey_st, khval_st) \
    typedef struct                             \
    {                                          \
        khint_st    m_buckets;  /* comments */ \
        khint_st    m_size;     /* comments */ \
        khint_st    m_occupied;                \
        khint_st    m_upper_bound;             \
        khint32_st *m_flags;    /* comments */ \
        khkey_st   *m_keys;                    \
        khval_st   *m_vals;                    \
        } kh_##name##_st;

// VS.

#define __KHASH_TYPE(name, khkey_st, khval_st) \
    typedef struct \
    { \
        khint_st    m_buckets; \
        khint_st    m_size; \
        khint_st    m_occupied; \
        khint_st    m_upper_bound; \
        khint32_st *m_flags; \
        khkey_st   *m_keys; \
        khval_st   *m_vals; \
        } kh_##name##_st;

// for readability, this is also make sense
#define A_MACRO         something
#define ANOTHER_MACRO   another_thing
#define YET_ALSO_MACRO  yet_also_something

// if it is to long to fit one line, breaking up like this
#define A_VERY_LONG_MACRO_NAME \
    a_good_idea_to_have_this_macro_so_long
```

Feel free to insert extra parentheses or braces judiciously:

- Maybe it is necessarily to make sure the code work correctly
- Maybe it will very helpful in increasing readability

Warning: If you can avoid using macros, just do not use them.

Vertical Whitespace

- Minimize use of vertical whitespace.
- Do not end functions with blank lines.
- Do not start functions with blank lines.
- Do not use blank lines when you do not have to.
- Do not put more than one or two blank lines between functions.
- Blank lines inside a chain of if-else blocks may well help readability.
- Blank lines at the beginning or end of a function very rarely help readability.

Tip: The more code that fits on one screen, the easier it is to follow and understand the control flow of the program. Of course, readability can suffer from code being too dense as well as too spread out, so use your judgment. But in general, minimize use of vertical whitespace.

Exceptions

You may diverge from the rules when dealing with code that does not conform to this style guide.

If you find yourself modifying code that was written to specifications other than those presented by this guide, you may have to diverge from these rules in order to stay consistent with the local conventions in that code. If you are in doubt about how to do this, ask the original author or the person currently responsible for the code.

Remember that **consistency** includes **local consistency**, too.

Ending

- Use **common** sense and be **consistency**.
- Take a few minutes to look at the code around you and determine its style.

Enough writing about writing code. Have fun, enjoy coding!

- [1] Google C++ Style Guide
- [2] Linux Kernel Coding Style
- [3] C Coding Standard
- [4] Recommended C Style and Coding Standards
- [5] SEI CERT C Coding Standard

CHAPTER 2

C++

Background

As every C++ programmer knows, the language has many powerful features, but this power brings with it complexity, which in turn can make code more bug-prone and harder to read and maintain.

The goal of this guide is to manage this complexity by describing in detail the dos and don'ts of writing C++ code. These rules exist to keep the code base manageable while still allowing coders to use C++ language features productively. And if you following the C part of CodingArt, then all will be consistent when you have to use C in C++, and no collision.

Style, also known as readability, is what called the conventions that govern the C++ code. The term `style` is a bit of a misnomer, since these conventions cover far more than just source file formatting.

Header Files

In general, every `.cc` file should have an associated `.h` file, but common exceptions are for **unittests** and small `.cc` files containing just a `main()` function.

Correct use of header files can make a huge difference to the readability, size and performance of your code.

Self Contained Headers

Header files should be self-contained (compile on their own) and end in `.h`.

All header files should be self-contained. Specifically, a header should have *header guards* and include all other headers it needs.

Non-header files that are meant for inclusion should end in `.inc` and be used sparingly. These are typically intended to be included at unusual locations, such as the middle of another file. They might not use *header guards*, and might not include their prerequisites. So, prefer self-contained headers when possible.

Prefer placing the definitions for **template** and **inline** functions in the same file as their declarations. The definitions of these constructs must be included into every `.cc` file that uses them, or the program may fail to link in some build configurations. If declarations and definitions are in different files, including the former should transitively include the latter. Do not move these definitions to separately included header files will make it simple.

As an exception, a template that is explicitly instantiated for all relevant sets of template arguments, or that is a private implementation detail of a class, is allowed to be defined in the one and only `.cc` file that instantiates the template.

The Header Guard

All header files should have `#define` guards to prevent multiple inclusion. The format of the symbol name should be `<PROJECT>_<PATH>_<FILE>_H`.

The `#define` guards should be uniqueness, based on the full path in a project's source tree. For example, the file `foo/bar/baz.h` in project `foo` should have the following guard:

```
/// @file
/// file description

#ifndef FOO_BAR_BAZ_H
#define FOO_BAR_BAZ_H
...
#endif // FOO_BAR_BAZ_H
```

Forward Declarations

Avoid using forward declarations where possible. Just `#include` the headers you need.

- Try to avoid forward declarations of entities defined in another project.
- When using a function declared in a header file, always `#include` that header.

Please see *Names and Order of Includes* for rules about when and where to `include` a header.

Names and Order of Includes

Use standard order for readability and to avoid hidden dependencies:

- C system headers
- C++ system headers
- dependency libraries headers
- your project headers

All of a project's header files should be listed as descendants of the project's source directory without use of UNIX directory shortcuts: the current directory(.) or the parent directory(..)

For example, the file `foo/bar/baz.h` in project `foo` should be include as:

```
#include "bar/baz.h"
```

Tip: Sometimes, system-specific code needs conditional includes. Such code can put conditional includes after other includes. Of course, keep your system-specific code small and localized.

Inline Functions

Define functions inline only when they are small, say, 10 lines or fewer.

Defination:

> You can declare functions in a way that allows the compiler to expand them inline rather than calling them through the usual function call mechanism.

Pros:

- Inlining a function can generate more efficient object code, as long as the inlined function is small.
- Feel free to inline accessors and mutators, and other short, performance-critical functions.

Cons:

- Overuse of inlining can actually make programs slower.
- Depending on a function's size, inlining it can cause the code size to increase or decrease.
- Inlining a very small accessor function will usually decrease code size while inlining a very large function can dramatically increase code size.

Tip:

- not inline a function if it is more than 10 lines long.
- it's typically not cost effective to inline functions with loops or switch statements.
- It's important to know that functions are not always inlined even if declared as such.

Header Constants

Do not use macros to define constants in headers, use enum instead if possible.

Scoping

Named Namespaces

- With few exceptions, always place code in a `namespace`.
- Do not use `using` directives, e.g. `using namespace foo;`.
- Do not declare anything in namespace `std`.
- Make proper use of `inline namespaces`.

Namespaces subdivide the global scope into distinct, named scopes, and so are useful for preventing name collisions in the global scope while allowing most code to use reasonably short names. Namespaces should have unique names based on the project name, and possibly its path.

Declaring entities in namespace `std` is undefined behavior, i.e., not portable. To declare entities from the standard library, just include the appropriate header file.

Use a `using` directive make all names from a namespace available pollutes the namespace.

```
// NEVER do it like this. This pollutes the namespace.
using namespace foo;
```

Do not use namespace **aliases** at namespace scope in header files except in explicitly marked internal-only namespaces, because anything imported into a namespace in a header file becomes part of the public API exported by that file.

Namespaces should be used as follows:

- follow *Namespace Names Naming Rules*.
- Terminate namespaces with comments as shown in the following example.
- Namespaces wrap the entire source file after includes.

```
// In header file: .h
//
// Notice:
// the open curly of namespace on the same line with the keyword, becase they are special.
namespace mynamespace {
```

```
// All declarations are within the namespace scope.
//
// Notice:
// the lack of indentation for code in the namespace make sense.
class MyClass
{
// Notice: the lack of indentation for public
public:
    // 4 spaces indent make it more readable.
    ...
    void foo(void);
};

  // It is clear enough where the namespace ends and which one.
} // namespace: mynamespace

////////////////////////////////////////////////////////////////////////////

// In source file: .cc
//
// Notice:
// the open curly of namespace on the same line with the keyword, becase they are special.
namespace mynamespace {

// Definition of functions is within scope of the namespace.
//
// Notice:
// the lack of indentation makes code not go that right far.
void MyClass::foo(void)
{
    do_some_thing();
}

} // namespace: mynamespace
```

Unnamed Namespaces

When definitions in a `.cc` file do **not** need to be referenced outside that file, place them in an unnamed namespace, and do not use them in the header files. Also see *Static Variables*.

All declarations can be given internal linkage by placing them in unnamed namespaces, and functions and variables can be given internal linkage by declaring them `static`. This means that anything you're declaring can not be accessed from another file. If a different file declares something with the same name, then the two entities are completely independent.

Format of unnamed namespaces like named namespaces. In the terminating comment, just leave the namespace name empty, e.g.

```
namespace {

do_some_thing_here
...

} // namespace:
```

Static Variables

When variables definied in a `.cc` file do not need to be referenced outside that file, declare them as `static`. It is rarely to define `static` variables in header files. Also see *Unnamed Namespaces*.

Scoped Functions

Functions in C++ have 2 type: **nonmember-functions** and **member-functions**.

- Prefer placing nonmember functions in a namespace, and rarely use global functions.
- Prefer grouping functions with a namespace instead of using a class as if it were a namespace.
- If you define a nonmember function and it is only needed in the `.cc` file, use internal linkage to limit its scope, which is use `static` keywords.
- Static member functions of a class should generally be closely related to instances of the class or the class's static data.

Nonmember functions and static member functions can be useful in some situations. Putting nonmember functions in a namespace avoids polluting the global namespace.

Sometimes it is useful to define a function not bound to a class instance. Such a function can be either a static member or a nonmember function. Nonmember functions should not depend on external variables, and should nearly always exist in a namespace. Rather than creating classes only to group static member functions which do not share static data, use namespaces instead, for example:

```
namespace myproject {
namespace foobar {
void function_1();
void function_2();

} // namespace: foobar
} // namespace: myproject
```

Instead of

```
namespace myproject {
class FooBar
{
public:
    static void function1();
    static void function2();
};
} // namespace: myproject
```

Static Storage Duration

Objects with static storage duration, including **global** variables, **static** variables, **static class** member variables, and **function static** variables, must be Plain Old Data (POD): only `int`, `char`, `float`, `double`, or `pointers`, `arrays` or `structs` of POD.

Variables of class type with static storage duration are forbidden, because they can cause hard-to-find bugs due to indeterminate order of construction and destruction. However, such variables are allowed if they are **constexpr**: they have no dynamic initialization or destruction.

The order in which class constructors and initializers for static variables are called is only partially specified in C++ and can even change from build to build, which can cause bugs that are difficult to find. Therefore in addition to banning globals of class type, we do not allow non-local static variables to be initialized with the result of a function, unless that function (such as `getenv()`, or `getpid()`) does not itself depend on any other globals. However, a static POD variable within function scope may be initialized with the result of a function, since its initialization order is well-defined and does not occur until control passes through its declaration.

Likewise, global and static variables are destroyed when the program terminates, regardless of whether the termination is by returning from `main()` or by calling `exit()`. The order in which destructors are called is defined to be the reverse of the order in which the constructors were called. Since constructor order is indeterminate, so is destructor order. One way to alleviate the destructor problem is to terminate the program by calling `quick_exit()` instead of `exit()`.

If you need a static or global variable of a class type, consider initializing a pointer (which will never be freed), from either your `main()` function or from `pthread_once()`. Note that this must be a raw pointer, not a **smart** pointer, since the smart pointer's destructor will have the order-of-destructor issue that we are trying to avoid.

Global Variables

Avoid using global variable where possible.

Locale Variables

Place a function's variables in the narrowest scope, and initialize variables in the declaration.

- Declare local variables in as local a scope as possible.
- Declare local variables as close to the first use as possible.

This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialized to. In particular, initialization should be used instead of declaration and assignment, e.g.

```
int i;
i = f();      // Bad  -- initialization separate from declaration.

int j = g(); // Good -- declaration has initialization.

// Bad - Prefer initializing using brace initialization.
std::vector<int> v;
v.push_back(1);
v.push_back(2);

// Good - v starts initialized.
std::vector<int> v = {1, 2};
```

Classes

Classes are the fundamental unit of code in C++.

Constructors

It is possible to perform arbitrary initialization in the body of the constructor.

- Avoid virtual method calls in constructors.
- Avoid initialization that can fail if you can't signal an error.
- No need to worry about whether the class has been initialized or not.
- Objects that are fully initialized by constructor call can be const and may also be easier to use with standard containers or algorithms.

Implicit Conversions

Do not define implicit conversions. Use the `explicit` keyword for conversion operators and single-argument constructors.

Implicit conversions allow an object of one type (called the source type) to be used where a different type (called the destination type) is expected, e.g. when passing an `int` argument to a function that takes a `double` parameter.

In addition to the implicit conversions defined by the language, users can define their own, by adding appropriate members to the class definition of the source or destination type. An implicit conversion in the source type

is defined by a type conversion operator named after the destination type (e.g. operator bool()). An implicit conversion in the destination type is defined by a constructor that can take the source type as its only argument (or only argument with no default value).

The `explicit` keyword can be applied to a constructor or (since C++11) a conversion operator, to ensure that it can only be used when the destination type is explicit at the point of use, e.g. with a `cast`. This applies not only to implicit conversions, but to C++11's list initialization syntax.

- Implicit conversions can make a type more usable and expressive by eliminating the need to explicitly name a type when it's obvious.
- Implicit conversions can be a simpler alternative to overloading.
- List initialization syntax is a concise and expressive way of initializing objects.
- Implicit conversions can hide type-mismatch bugs, where the destination type does not match the user's expectation, or the user is unaware that any conversion will take place.
- Implicit conversions can make code harder to read, particularly in the presence of overloading, by making it less obvious what code is actually getting called.
- Constructors that take a single argument may accidentally be usable as implicit type conversions, even if they are not intended to do so.
- When a single-argument constructor is not marked explicit, there's no reliable way to tell whether it's intended to define an implicit conversion, or the author simply forgot to mark it.
- It's not always clear which type should provide the conversion, and if they both do, the code becomes ambiguous.
- List initialization can suffer from the same problems if the destination type is implicit, particularly if the list has only a single element.

Type conversion operators, and constructors that are callable with a single argument, must be marked `explicit` in the class definition. As an exception, copy and move constructors should not be explicit, since they do not perform type conversion. Implicit conversions can sometimes be necessary and appropriate for types that are designed to transparently wrap other types. In that case, contact your project leads to request a waiver of this rule.

Constructors that cannot be called with a single argument should usually omit explicit. Constructors that take a single `std::initializer_list` parameter should also omit explicit, in order to support copy-initialization (e.g. `MyType m = {1, 2};`).

Copyable and Movable Types

Support copying and/or moving if these operations are clear and meaningful for your type. Otherwise, disable the implicitly generated special functions that perform copies and moves.

A **copyable** type allows its objects to be initialized or assigned from any other object of the same type, without changing the value of the source. For user-defined types, the copy behavior is defined by the copy constructor and the copy-assignment operator. `string` is an example of a copyable type.

A **movable** type is one that can be initialized and assigned from temporaries (all copyable types are therefore movable). `std::unique_ptr<int>` is an example of a movable but not copyable type. For user-defined types, the move behavior is defined by the move constructor and the move-assignment operator.

The copy/move constructors can be implicitly invoked by the compiler in some situations, e.g. when passing objects by value.

Objects of copyable and movable types can be passed and returned by value, which makes APIs simpler, safer, and more general. Unlike when passing objects by pointer or reference, there's no risk of confusion over ownership, lifetime, mutability, and similar issues, and no need to specify them in the contract. It also prevents non-local interactions between the client and the implementation, which makes them easier to understand, maintain, and optimize by the compiler. Further, such objects can be used with generic APIs that require pass-by-value, such as most containers, and they allow for additional flexibility in e.g., type composition.

Copy/move constructors and assignment operators are usually easier to define correctly than alternatives like `Clone()`, `CopyFrom()` or `Swap()`, because they can be generated by the compiler, either implicitly or with = default. They are concise, and ensure that all data members are copied. Copy and move constructors are also generally more efficient, because they don't require heap allocation or separate initialization and assignment steps, and they're eligible for optimizations such as copy elision.

Move operations allow the implicit and efficient transfer of resources out of rvalue objects. This allows a plainer coding style in some cases.

Some types do not need to be copyable, and providing copy operations for such types can be confusing, nonsensical, or outright incorrect. Types representing singleton objects (Registerer), objects tied to a specific scope (Cleanup), or closely coupled to object identity (Mutex) cannot be copied meaningfully. Copy operations for base class types that are to be used polymorphically are hazardous, because use of them can lead to object slicing. Defaulted or carelessly-implemented copy operations can be incorrect, and the resulting bugs can be confusing and difficult to diagnose.

Copy constructors are invoked implicitly, which makes the invocation easy to miss. This may cause confusion for programmers used to languages where pass-by-reference is conventional or mandatory. It may also encourage excessive copying, which can cause performance problems.

Provide the copy and move operations if their meaning is clear to a casual user and the copying/moving does not incur unexpected costs. If you define a copy or move constructor, define the corresponding assignment operator, and vice-versa. If your type is copyable, do not define move operations unless they are significantly more efficient than the corresponding copy operations. If your type is not copyable, but the correctness of a move is obvious to users of the type, you may make the type move-only by defining both of the move operations.

If your type provides copy operations, it is recommended that you design your class so that the default implementation of those operations is correct. Remember to review the correctness of any defaulted operations as you would any other code, and to document that your class is copyable and/or cheaply movable if that's an API guarantee.

Structs VS. Classes

Use a `struct` only for passive objects that carry data, and everything else is a `class`.

The `struct` and `class` keywords behave almost identically in C++. We add our own semantic meanings to each keyword, so you should use the appropriate keyword for the data-type you're defining.

`structs` should be used for passive objects that carry data, and may have associated constants, but lack any functionality other than access/setting the data members. The accessing/setting of fields is done by directly accessing the fields rather than through method invocations. Methods should not provide behavior but should only be used to set up the data members, e.g., constructor, destructor, Initialize(), Reset(), Validate().

If more functionality is required, a `class` is more appropriate. If in doubt, make it a `class`.

For consistency with STL, you can use `struct` instead of `class` for functors and traits.

Note that member variables in `struct` and `class` have different naming rules. Data members are all lowercase, with underscores between words, can have digit number if necessary. Data members of `class` leading with `x_`, while data members of `struct` have perfix of `m_`, e.g.

```
// here we use suffix of _st, because in C++ we can omit struct
// directly use the struct name, not like C way.
struct a_struct_type_st
{
    int m_age;
};

class ThisIsClass
{
    string x_name;
};
```

Inheritance

Composition is often more appropriate than inheritance. When using inheritance, make it public.

Multiple Inheritance

Only very rarely is multiple implementation inheritance actually useful. We allow multiple inheritance only when at most one of the base classes has an implementation; all other base classes must be pure interface classes tagged with the Interface suffix.

Interfaces

Classes that satisfy certain conditions are allowed, but not required, to end with an Interface suffix.

Operator Overloading

Overload operators judiciously. Do not create user-defined literals.

Access Control

Make data members private, unless they are `static const`.

Declaration Order

Group similar declarations together, placing `public` parts earlier.

A class definition should usually start with a `public` section, followed by `protected`, then `private`. Omit sections that would be empty.

Within each section, generally prefer grouping similar kinds of declarations together, and generally prefer the following order:

- types (including `typedef`, `using`, and nested `struct` and `class`)
- constants
- factory functions
- constructors
- assignment operators
- destructor
- all other methods
- data members

Do not put large method definitions `inline` in the class definition. Usually, only trivial or performance-critical, and very short, methods may be defined `inline`.

Functions

Write Short Functions

Prefer small and focused functions, it is more readable and more manageable.

We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Another measure of the function is the number of local variables. They shouldnâĂŹt exceed 5-10. If not, then re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know youâĂŹre brilliant, but maybe youâĂŹd like to understand what you did 2 weeks from now.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

Parameter Ordering

When defining a function, parameter order is: `inputs`, then `outputs`.

Parameters to C/C++ functions are either input to the function, output from the function, or both. Input parameters are usually values or const references, while output and input/output parameters will be pointers to non-const. When ordering function parameters, put all input-only parameters before any output parameters. In particular, do not add new parameters to the end of the function just because they are new; place new input-only parameters before the output parameters.

This is not a hard-and-fast rule. Parameters that are both input and output (often classes/structs) muddy the waters, and, as always, consistency with related functions may require you to bend the rule.

Reference Arguments

All parameters passed by reference must be labeled const.

In C, if a function needs to modify a variable, the parameter must use a pointer, e.g.

```
int foo(int *pval);
```

In C++, the function can alternatively declare a reference parameter, e.g.

```
int foo(int &val);
```

Static Functions

When functions defined in a `.c` file, and will never be used outside of that file, then just declare them as `static`.

Function Overloading

Use overloaded functions (including constructors) only if a reader looking at a call site can get a good idea of what is happening without having to first figure out exactly which overload is being called.

Default Arguments

Default arguments are allowed on non-virtual functions when the default is guaranteed to always have the same value.

Trailing Return Type Syntax

Use trailing return types only where using the ordinary syntax (leading return types) is impractical or much less readable.

C++ Features

Rvalue References

Use rvalue references only to define move constructors and move assignment operators, or for perfect forwarding.

Friends

We allow use of friend classes and functions, within reason.

Friends should usually be defined in the same file so that the reader does not have to look in another file to find uses of the private members of a class. A common use of friend is to have a FooBuilder class be a friend of Foo so that it can construct the inner state of Foo correctly, without exposing this state to the world. In some cases it may be useful to make a unittest class a friend of the class it tests.

Friends extend, but do not break, the encapsulation boundary of a class. In some cases this is better than making a member public when you want to give only one other class access to it. However, most classes should interact with other classes solely through their public members.

Exceptions

We do not use C++ exceptions.

Run-Time Type Information (RTTI)

Avoid using Run Time Type Information (RTTI).

Casting

todo

Streams

todo

```
int  is_good = 1;     // Bad  - should have type bool.
bool is_good = true;  // Good - just use boolean type where possible.
```

Variable Declarations

Declare only one variable per line, and each line have only one sentence.

```
int i, j = 1;      // never do this
int k=0; func();   // never do this also

int i = 0;         // this will more clear
int j = 1;         // no bug easy to hide
```

```
int k = 0;
func();
```

0 and NULL

- Use `0` for integers;
- Use `0.0` for reals;
- Use `NULL` for pointers;
- Use `'\0'` for chars;

Usage of sizeof

Prefer `sizeof(varname)` to `sizeof(type)`.

Use `sizeof(varname)` when you take the size of a particular variable. `sizeof(varname)` will update appropriately if someone changes the variable type either now or later.

You may use `sizeof(type)` for code unrelated to any particular variable.

Usage of goto

Just do not use `goto` when it is absolutely necessary.

The goto statement comes in handy when a function exits from multiple locations and some common work such as cleanup has to be done. If there is no cleanup needed then just return directly.

The use of `goto` make code hard to read and management, so just use it as rare as possible. If for some reason, you must use `goto`, then choose label names which say what the goto does or why the goto exists.

The rationale for using gotos is:

- nesting is reduced.
- errors by not updating individual exit points when making modifications are prevented.
- saves the compiler work to optimize redundant code away.

A common type of bug to be aware of is one `err` bugs which look like this:

```
err:
    kfree(foo->bar);
    kfree(foo);
    return ret;
```

The bug in this code is that on some exit paths `foo` is `NULL`. Normally the fix for this is to split it up into two error labels `err_free_bar` and `err_free_foo`, e.g.

```
err_free_bar:
    kfree(foo->bar);
err_free_foo:
    kfree(foo);
    return ret;
```

Usage of Macros

Macros with multiple statements should be enclosed in a `do-while` block, e.g.

```
// make properly align of \'s to increase the readability
#define macrofun(a, b, c)    \
    do                       \
    {                        \
        if(a == 5)           \
        {                    \
            do_this(b, c);   \
        }                    \
    }while(0)
```

- Avoid using macros if they affect control flow, e.g.

```
#define FOO(x)                  \
    do                          \
    {                           \
        if(blah(x) < 0)         \
        {                       \
            return ERROR_CODE;  \
        }                       \
    }while(0)
```

- Avoid using macros if they depend on having a local variable with a magic name, e.g.

```
// what the hell of them?
#define FOO(val) bar(index, val)
```

- Make the expression precedence very very clear by using properly parentheses.

Conditional Compilation

Wherever possible, donâĂŹt use preprocessor conditionals (`#if`, `#ifdef`, etc.) in `.c` files, and doing so makes code harder to read and logic harder to follow. Instead, use such conditionals in a header file defining functions for use in those `.c` files, providing no-operation stub versions in the `#else` case, and then call those functions unconditionally from `.c` files. The compiler will avoid generating any code for the stub calls, producing identical results, but the logic will remain easy to follow.

If you have a function or variable which may potentially go unused in a particular configuration, and the compiler would warn about its definition going unused, so just mark the definition as `__attribute__((unused))` (see See GCC Attribute Syntax) rather than wrapping it in a preprocessor conditional. However, if a function or variable always goes unused, then just delete it.

At the end of any non-trivial `#if` or `#ifdef` block (more than a few lines), place a comment after the `#endif` on the same line, noting the conditional expression used. For instance:

```
#ifdef CONFIG_SOMETHING

doing_some_thing

#endif // CONFIG_SOMETHING
```

Also do NOT not put `#ifdef` in an expressions for readability.

Naming

The most important consistency rules are those that govern naming. The style of a name immediately informs us what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc., without requiring us to search for the declaration of that entity. The pattern-matching engine in our brains relies a great deal on these naming rules.

Naming rules are pretty arbitrary, but we feel that consistency is more important than individual preferences in this area, so regardless of whether you find them sensible or not, the rules are the rules.

General Naming Rules

- Use intention-revealing names, names should reveal real intent, avoid disinformation.
- Make names should have meaningful distinctions and try best make them pronounceable.
- Make it clear, easy to read, write and understand, within reason.
- Names should be descriptive, avoid abbreviation.

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader.

Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do NOT abbreviate by deleting letters within a word. Note that certain universally-known abbreviations are OK, such as i for an iteration variable.

```
int price_count_reader;    ///< Good - No abbreviation.
int num_errors;            ///< Good - "num" is a widespread convention.
int num_dns_connections;   ///< Good - Most people know what "DNS" stands for.

int n;                     ///< Bad  - Meaningless.
int nerr;                  ///< Bad  - Ambiguous abbreviation.
int n_comp_conns;          ///< Bad  - Ambiguous abbreviation.
int wgc_connections;       ///< Bad  - Only your group knows what this stands for.
int pc_reader;             ///< Bad  - Lots of things can be abbreviated "pc".
int cstmr_id;              ///< Bad  - Deletes internal letters.
```

File Naming Rules

Filenames should be all lowercase with underscores between words, if necessary can have digital numbers. C files should end in `.c` and header files should end in `.h`.

Use underscores to separate words. Examples of acceptable file names:

- my_useful_file.c
- my_useful_file.h

Tip:

- Do not use filenames that already exist in `/usr/include`, such as `db.h`.
- In general, make your filenames very specific. For example, use `http_server_logs.h` rather than `logs.h`.

Namespace Names Naming Rules

- Namespace names are all lower-case, without underscore and digital number.
- Top-level namespace names are based on the project name.
- Avoid collisions between nested namespaces and well-known top-level namespaces.

The name of a top-level namespace should usually be the name of the project or team. The code in that namespace should usually be in a directory whose basename matches the namespace name.

Keep in mind that the rule against abbreviated names applies to namespaces just as much as variable names. Code inside the namespace seldom needs to mention the namespace name, so there's usually no particular need for abbreviation anyway.

Avoid nested namespaces that match well-known top-level namespaces. Collisions between namespace names can lead to surprising build breaks because of name lookup rules. In particular, do not create any nested `std` namespaces.

For internal namespaces, be wary of other code being added to the same internal namespace causing a collision. In such a situation, using the filename to make a unique internal name is helpful, for example: `foo::helper::message_internal` for use in `message.h` header file.

Type Naming Rules

Typedef-ed types should be all lowercase with underscores between words, and have one of suffix:

- suffix `_st` for `typedef struct`
- suffix `_et` for `typedef enum`
- suffix `_ut` for `typedef union`
- suffix `_bt` for basic type aliases, e.g. `typedef int buffer_id_bt;`
- suffix `_ft` for function type, e.g. `typedef int (my_func_ft)(int cnt);`

Non-Typedef-ed `struct`/`enum`/`union` for forward necessary declaration, with one of suffix as following:

- suffix `_ST` for `struct`
- suffix `_ET` for `enum`
- suffix `_UT` for `union`

Tip: For convenience, it is recommended to typedef user defined types, and all use typedef version.

Common Variable Naming Rules

All variable names consist of lowercase and underscores, if necessary can have digital numbers.

For example:

```
string table_name;  ///< OK  - uses underscore.
string tablename;   ///< OK  - all lowercase.

string tableName;   ///< Bad - mixed case.
```

Tip:

- It maybe a good idea to make and use searchable names.

Struct Member Naming Rules

Members of struct are named like *common variables* with prefix `m_`.

Enum Member Naming Rules

Members of enum are named like *common variables* with prefix `k_`.

Tip: It maybe a good idea to hava format like, `k_<id>`, where `id` is a short name derived from that enumeration.

Union Member Naming Rules

Members of union are named like *common variables* with prefix `m_`.

Global Variable Naming Rules

Global variable name just like *common variables*, but with prefix `g_`.

Static Variable Naming Rules

Static variable name just like *common variables*, but with prefix `s_`.

Local Variable Naming Rules

Local variable just following *Common Variable Naming Rules*.

Local variable names should be short, and to the point. If you have some random integer loop counter, it should probably be called `i`. It is clear enough and there is no chance of it being mis-understood. Similarly, `tmp` can be just about any type of variable that is used to hold a temporary value.

Constant Variable Naming Rules

All constant variables, and whose value is fixed for the duration of the program, following *Common Variable Naming Rules*, but with a leading `k`. Also see *Enum Member Naming Rules*.

Function Naming Rules

Function names consist of lowercase and underscores, if necessary can have digital numbers.

Tip: It maybe a good idea to have a perfix for a serial or module of functions.

Macro Naming Rules

Macro names consist of uppercase and underscores, if necessary can have digital numbers.

- If macros are resembling functions, then name them in lower case is better.
- If a macros can be empty, then always use capitalized letters, e.g. `DEBUG_MSG(msg)`.

```
/// header file guard macro
#define <PROJECT>_<PATH>_<FILE>_H

/// awesome macro defination
#define AWESOME_MACRO_DEFINATION

/// constant number value
#define PI (3.1415926)

/// constant string value
#define CONFIG_FILE_NAME "config"

/// function like macro
#ifdef SHOW_DEBUG_MESSAGE
#   define DEBUG_MSG(msg) printf("%s\n", msg);
```

```
#else
#   define DEBUG_MSG(msg)
#endif
```

Note:

- General speaking, if not necessary, macros should not be used.

Comments

Though a pain to write, comments are absolutely vital to keeping our code readable. The following rules describe what you should comment and where.

If we make a step further, just do little work while writing comments, then you can use the doc-tools out there to automatically generating perfect documentation. Here we use the well known documentation tool `doxygen`.

Tip:

- While comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that you must then explain through comments.
- do not abuse comments. Comments are good, but there is also a danger of over-commenting. NEVER try to explain HOW your code works in a comment: itâĂŹs much better to write the code so that the working is obvious, and itâĂŹs a waste of time to explain badly written code.

Note: Be generous - the next one may be you!

When writing comments, write for your audience: the next contributor who will need to understand your code.

Comment Style

Prefer using the //-style syntax only, much more easier to type.

```
// This is a comment spanning
// multiple lines
func();
```

Tip:

- Use either the // or /* */ syntax, as long as you are consistent.

File Comments

Every file should have a comment at the top describing its contents. For example:

```
/// @file
/// A brief description of this file.
///
/// A longer description of this file.
/// Be very generous here.
```

Tip: Generally speaking:

- Comments in `.h` file will describe the variables and functions that are declared in the file with an overview of what they are for and how they are used.
- Comments in `.c` file should contain more information about implementation details or discussions of tricky algorithms. If you feel the implementation details or a discussion of the algorithms would be useful for someone reading the `.h`, feel free to put it there instead, but mention in the `.c` that the documentation is in the `.h` file.

Warning: Do not duplicate comments in both the `.h` and the `.c`. Duplicated comments diverge.

Structured Data Comments

Every `struct`, `enum` and `union` definition should have accompanying comments that describes what it is for and how it should be used.

If the field comments are short, you can put them next to the field, for example:

```
/// Structure used for growing arrays.
///
/// This is used to store information that only grows, and deleted all at once,
/// and needs to be accessed by index. Also see @ref ga_clear() and @ref ga_grow().
typedef struct growarray
{
    int   m_ga_size;        ///< current number of items used
    int   m_ga_maxsize;     ///< maximum number of items possible
    int   m_ga_itemsize;    ///< sizeof(item)
    int   m_ga_growsize;    ///< number of items to grow each time
    void *m_ga_data;        ///< pointer to the first item
}garray_st;
```

If the field comments are long, you can put them previous to the field, for example:

```
/// ...
typedef struct growarray
{
    /// current number of items used
    int   m_ga_size;
    /// maximum number of items possible
    int   m_ga_maxsize;
    /// sizeof(item), item size in bytes
    int   m_ga_itemsize;
    /// number of items to grow each time
    int   m_ga_growsize;
    /// pointer to the first item
    void *m_ga_data;
}garray_st;
```

Function Declarations Comments

Comments at the declaration of a function describe the *usage* of the function. Every function declaration should have comments immediately preceding it that describe what the function does and how to use it. In general, these comments do not describe how the function performs its task which should be left to comments in the function definition.

Types of things to mention in comments at the function declaration:

- If the function allocates memory that the caller must free.
- Whether any of the arguments can be a null pointer.

- If there are any performance implications of how a function is used.
- Whether the function is re-entrant.
- What are its synchronization assumptions.

```
/// Brief description of the function.
///
/// Detailed description.
/// May span multiple paragraphs.
///
/// @param[in] arg1 Description of arg1
/// @param[in] arg2 Description of arg2. May span
///                 multiple lines.
///
/// @return Description of the return value.
iterator_st *get_iterator(void *arg1, void *arg2);
```

Function Definition Comments

Comments at the definition of a function describe *operation* of the function. If there is anything tricky about how a function does its job, the function definition should have an explanatory comment. For example, in the definition comment you might describe any coding tricks you use, give an overview of the steps you go through, or explain why you chose to implement the function in the way you did rather than using a viable alternative. For instance, you might mention why it must acquire a lock for the first half of the function but why it is not needed for the second half.

```
// Note that do not use Doxygen comments here. They are not for Doxygen.
iterator_st *get_iterator(void *arg1, void *arg2)
{
...
}
```

Note: Do not just repeat the comments given with the function declaration, in the `.h` file or wherever. It's okay to recapitulate briefly what the function does, but the focus of the comments should be on how it does it.

Variable Comments

In general the actual name of the variable should be descriptive enough to give a good idea of what the variable is used for. In certain cases, more comments are required.

```
/// The total number of tests cases that we run through in this regression test.
const int g_test_cases_num = 6;
```

Note:

- All global variables should have a comment describing what they are and what they are used for.

Implementation Comments

In your implementation you should have comments in tricky, non-obvious, interesting, or important parts of your code.

Explanatory Comments: tricky or complicated code blocks should have comments before them.

```
// Divide result by two, taking into account that x contains the carry from the add.
for(int i = 0; i < result->m_size; i++)
{
    x = (x << 8) + (*result)[i];
    (*result)[i] = x >> 1;
    x &= 1;
}
```

Line Comments: lines that are non-obvious should get a comment at the end of the line. These end-of-line comments should be separated from the code using spaces.

```
// If we have enough memory, mmap the data portion too.
mmap_budget = max<int64>(0, mmap_budget - index->m_length);
if(mmap_budget >= data_size && !map_data(mmap_chunk_bytes, mlock))
{
    return;  // Error already logged.
}
```

Line Up Comments: if you have several comments on subsequent lines, it can often be more readable to line them up:

```
do_something();                         // Comment here so the comments line up.
do_something_else_that_is_longer();     // Comment here so there are two spaces between
                                        // the code and the comment.
{
    do_something_else();                // Comment here so the comments line up.
}
```

No Magic Arguments: when you pass in a null pointer, boolean, or literal integer values to functions, you should consider adding a comment about what they are, or make your code self-documenting by using constants. For example, compare:

```
bool success = calculate_something(interesting_value,
                                   10,     // What is this ?
                                   false,  // What is this ?
                                   NULL);  // What is this ?
```

versus:

```
bool success = calculate_something(interesting_value,
                                   10,     // Default base value.
                                   false,  // Not the first time we're calling this.
                                   NULL);  // No callback.
```

Or alternatively, constants or self-describing variables:

```
// line them up make more readable, both definition and comments
const int    default_base_value = 10;     // Default base value.
const bool   first_time_calling = false;  // Not the first time calling this.
callback_ft  null_callback = NULL;        // No callback

bool success = calculate_something(interesting_value,
                                   default_base_value,
                                   first_time_calling,
                                   null_callback);
```

Tip:

- Never describe the code itself, just assume the reader knows C better than you.
- Never abuse comment, do not state the obvious.
- Provide higher level comments that describe why the code does what it does.

Punctuation, Spelling and Grammar

Pay attention to punctuation, spelling, and grammar. It is easier to read well-written comments than badly written ones.

Comments should be as readable as narrative text, with proper capitalization and punctuation. In many cases, complete sentences are more readable than sentence fragments. Shorter comments, such as comments at the end of a line of code, can sometimes be less formal, but you should be consistent with your style.

Although it can be frustrating to have a code reviewer point out that you are using a comma when you should be using a semicolon, it is very important that source code maintain a high level of clarity and readability. Proper punctuation, spelling, and grammar help with that goal.

TODO Comments

Use TODO comment for code that is temporary, a short-term solution, or good-enough but not perfect.

TODO comment should include the string **TODO** or **todo**, followed by the name, e-mail address, bug ID, or other identifier(person or issue), which can provide best context about the problem referenced by the TODO Comment. The main purpose is to have a consistent TODO comment format that can be searched to find out how to get more details upon request. A TODO comment is not a commitment that the person referenced will fix the problem. Thus when you create a TODO comment with a name, it is almost always your name that is given.

If TODO comment is of the form: *at a future date do something*

- either include a very specific date, e.g. *Fix by November 2005*
- either include a very specific event, e.g. *Remove this code when all clients can handle XML responses.*

```
/// @todo (kl@gmail.com): Use a "*" here for concatenation operator.
/// @todo (Zeke): change this to use relations.
/// @todo (bug 12345): remove the "Last visitors" feature.

//  TODO (kl@gmail.com): Use a "*" here for concatenation operator.
//  TODO (Zeke): change this to use relations.
//  TODO (bug 12345): remove the "Last visitors" feature.
```

Deprecation Comments

Use Deprecation Comment for the interface API that is deprecated.

You can mark an interface as deprecated by writing a comment containing the word **DEPRECATED** or **deprecated**, followed by your name, e-mail address, or other identifier in parentheses. The comment goes either before the declaration of the interface or on the same line as the declaration.

A deprecation comment must include simple, clear directions for people to fix their callsites. In C, you can implement a deprecated function as an inline function that calls the new interface point.

Marking an interface point deprecated will not magically cause any callsites to change. If you want people to actually stop using the deprecated facility, you will have to fix the callsites yourself or recruit a crew to help you.

New code should not contain calls to deprecated interface points. Use the new interface point instead. If you cannot understand the directions, find the person who created the deprecation and ask them for help using the new interface point.

Formatting

Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some

getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

Line Length

Each line of text in your code should be at most 100 characters long.

Note:

- Maybe you are not agree with 100 as maxlength of lines, exactly is 99 visiable characters with one invisiable character(new line sign), and prefer to 80, 120 or others. Thus, regardless of whether you find them sensible or not, the rules are the rules.
- If you are writing for print using A4, changing this back to 80 maybe more reasonable, because the max character length of A4 is 80.
- Comment lines can be longer than 100 characters if it is not feasible to split them without harming readability. e.g. if a line contains an example command or a literal URL longer than 100 characters.
- A raw-string literal may have content that exceeds 100 characters.
- An `#include` statement with a long path may exceed 100 columns.

Indentation

Tabs are 4 characters, and thus indentations are also 4 characters, and use spaces only.

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

If having 4-character indentations makes the code move too far to the right, and makes it hard to read on the screen, then it maybe a warning that your logical have problems and you need to find an easy way to fix your problem.

Tip:

- DonâĂŹt put multiple statements on a single line unless you have something to hide.
- DonâĂŹt put multiple assignments on a single line either.
- Avoid complicated expressions.

Spaces VS. Tabs

Use only spaces, and indent 4 spaces at a time. Do not use tabs in your code.

Tip:

- You should set your editor to emit spaces when you hit the tab key.
- Put it this way will make the code format always correct with all text editors.

Warning: Use tabs when it is necessary, such as the Makefile rules.

Non-ASCII Characters

Non-ASCII characters should be rare, and must use UTF-8 formatting.

You shouldn't hard-code user-facing text in source, even English, so use of non-ASCII characters should be rare. However, in certain cases it is appropriate to include such words in your code. For example, if your code parses data files from foreign sources, it may be appropriate to hard-code the non-ASCII string(s) used in those data files as delimiters. More commonly, unittest code (which does not need to be localized) might contain non-ASCII strings. In such cases, you should use UTF-8, since that is an encoding understood by most tools able to handle more than just ASCII.

Hex encoding is also OK, and encouraged where it enhances readability, for example, `\uFEFF`, is the Unicode zero-width no-break space character, which would be invisible if included in the source as straight UTF-8.

Breaking Long Lines and Strings

Coding style is all about readability and maintainability using commonly available tools.

Statements longer than 100 columns will be broken into sensible chunks, unless exceeding 100 columns significantly increases readability and does not hide information. Descendants are always substantially shorter than the parent and are placed substantially to the right. However, never break user-visible strings messages, because that breaks the ability to `grep` for them.

The Usage of Braces

Put the opening and closing brace on the line just by itself, for all statement blocks, thusly:

```
if(is_true)
{
    do_it_like_this();
    // decrease the code density, make the start/end more clear
}
else if(is_good)
{
    do_some_thing();
}
else
{
    do_other_thing();
}

struct this_is_good
{
    bool m_good;
    ...
}
```

Note that the closing brace is empty on a line of its own, the only exception is it followed by by a continuation, that is a do-statement, e.g.

```
do
{
    do_it_like_this();
}while(is_true);
```

Prefer curly brace where a single statement is enough, make it clear enough, e.g:

```
if(condition)
{
    action();
}
```

```
if(condition)
{
    do_something();
}
else
{
    do_another();
}
```

Tip:

- clearness and readability is much more important.
- do not worried about saving lines.

The Usage of Spaces

- NO spaces after the keywords, the notable exceptions of C and the function names.
- NO spaces after-the-open and before-the-close parentheses.
- NO space around the `.` and `->` structure member operators.

```
// Keywords of C
if, switch, case, for, do, while

// Notable exceptions of C
sizeof, typeof, alignof, __attribute__
```

```
// do not need to emphasis the keywords, it is clear enough
while (condition)
{
    do_something();
}

// do not need to emphasis the condition, it is clear enough
if( condition )
{
    do_something();
}

s = sizeof( struct file ); // This is not good.
s = sizeof( struct file ); // This is good enough.
```

- Use one space around (on each side of) most binary and ternary operators, such as any of these:

```
=  +  -  <  >  *  /  %  |  &  ^  <=  >=  ==  !=  ?  :
```

- NO space after unary operators, such as any of these:

```
&  *  +  -  ~  !  sizeof  typeof  alignof  __attribute__  defined
```

- NO space before the postfix increment and decrement unary operators:

```
++  --
```

- NO space after the prefix increment and decrement unary operators:

```
++  --
```

Note: Although, for notable exceptions, the parentheses are not required in the language, for example, `sizeof info;` is the same as `sizeof(info);` after `struct fileinfo info;` is declared, it will make things simple by using parentheses all the time.

The Usage of Stars

When declaring pointer variable or a function that returns a pointer type, the preferred use of `*` is adjacent to the variable name or function name and not adjacent to the type name, e.g:

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

Function Declarations and Definitions

Return type on the same line as function name, parameters on the same line if they fit. Wrap parameter lists which do not fit on a single line.

Function on the same line, for example:

```
return_type function_name(type arg_name_1, type arg_name_2)
{
    do_something();
    ...
}
```

Function on more then one line, too much text to fit on one line, for example:

```
return_type function_name_1(type arg_name_1, type arg_name_2, type arg_name_3,
                            type arg_name_4)
{
    do_something();
    ...
}

return_type function_name_2(type arg_name_1, type arg_name_2, type arg_name_3,
                            type arg_name_4, type arg_name_5, type arg_name_6)
{
    do_something();
    ...
}
```

Note:

- Choose good parameter names.
- The open parenthesis is always on the same line as the function name.
- There is never a space between the function name and the open parenthesis.
- There is never a space between the open parentheses and the first parameters.
- The open curly brace is always on the next line by itself.
- The close curly brace is always on the last line by itself.
- All parameters should be named, with identical name in declaration and implementation.
- All parameters should be aligned if possible.
- Default indentation is 4 spaces.

- Wrapped parameters should indent to the function's first arguments.

Tip: Maybe it is time to rewrite the function interface by group the arguments into a struct if it has too much text to fit on one line.

Function Calls

Write the call all on a single line if it fits, function calls have the following format:

```
bool retval = do_something(arg_1, arg_2, arg_3);
```

If the arguments do not fit on one line, they should be broken up onto multiple lines, with each subsequent line aligned with the first argument. Do not add spaces after the open paren or before the close paren:

```
bool retval = do_something(a_very_very_very_very_long_arg_1,
                           arg_2, arg_3);
```

If the function has many arguments, consider having one per line if this makes the code more readable:

```
bool retval = do_something(arg_1,
                           arg_2,
                           arg_3,
                           arg_4);
```

If the function has many arguments, consider having minimum number of lines by breaking up onto multiple lines, with each subsequent line aligned with the functions's first argument:

```
bool retval = do_something(arg_1, arg_2, arg_3, arg_4
                           arg_5, arg_6, arg_7, arg_8);
```

Arguments may optionally all be placed on subsequent lines, with one line per argument:

```
if(...)
{
    do_something(arg_1,
                 arg_2,
                 arg_3,
                 arg_4);
}
```

Braced Initializer List

Format a braced list exactly like you would format a function call in its place.

If the braced list follows a name (e.g. a type or variable name), format as if the { } were the parentheses of a function call with that name. If there is no name, assume a zero-length name.

```
struct my_struct_ST m =
{
    superlongvariablename_1,
    superlongvariablename_2,
    { short, interior, list },
    {
        interiorwrappinglist_1,
        interiorwrappinglist_2,
    }
};
```

Conditionals

- Prefer no spaces inside parentheses.
- The `if`, `else` and `if else` keywords belong on separate lines by itself, no curly.
- Always use curly braces, even if the body is only one sentence.
- Make 4 space indent, make sure no use tabs.
- Make sure there is no space between `if`/`else`/`if else` keywords and the open parentheses.

```
// Good - no spaces inside parentheses
// Good - no spaces between if and the open parentheses
// Good - if just on the line by itself
if(condition)
{         // Good - open curly on the next line by itself
    ...   // Good - 4 space indent
}         // Good - close curly on the last line by itself
else if(...)
{
    ...
}
else
{
    ...
}

if( condition ) // Bad - have two spaces inside parentheses
{
  do_some();    // Bad - not 4 space indent
  ...
}
else if(...) {  // Bad - open curly and else-if not on the line just by itself
    ...
}
else {          // Bad - else/open curly not on the line just by itself
    ...
}
```

Even if the body is only one sentence, the curly can still not be omitted. Never use a single sentence or empty curly as the body, so the single semicolon.

```
if(x == foo) { return foo(); } // Good - this will be fine.
if(x == foo)
{
    return foo();              // Good - clear enough.
}

if(x == bar) bar();            // Bad  - this is not good, easy misreading
do_another_thing();

if(x == bar) return bar();     // Bad  - no curly.
if(x == bar) {}                // Bad  - do you realy need this?
```

Loops and Switch Statements

Empty loop bodies should only use an `continue` inside curly. Never use a single sentence or empty curly as the body, so the single semicolon.

```
while(condition) { continue; }          // Good - continue indicates no logic.
while(condition)
{
```

```
    continue;                               // Good - clear enough.
}

while(condition) {}                         // Bad  - is this part finished?
for(int i = 0; i < some_number; i++) {} // Bad  - why not do it in the body?
while(condition);                           // Bad  - looks like part of do/while loop.
```

- Single-statement loops should always have braces.

```
for(int i = 0; i < some_number; ++i)
{
    printf("I take it back\n");  // Good - 4 space indent
}

while(condition)
{
    do_something();               // Good - 4 space indent
}

for(int i = 0; i < some_number; ++i)
    printf("I love you\n");      // Bad   - no braces

for(int i = 0; i < some_number; ++i)
{
  printf("I take it back\n");    // Bad  - not 4 space indent
}
```

- `case` blocks in `switch` statements should always have curly braces.
- align the subordinate `case` labels in the same column with `switch`.
- `switch` statements should always have a `default` case, no exception.
- No space before the colon of `case`.
- If the `default` case should never execute, simply `assert`.

```
switch(var)
{           // open curly braces must on the next line by itself
case 0:     // each case must 4 space indent
{
    ...
    break;  // 4 space indent
}
case 1:     // no space before the colon
{
    ...
    break;
}
default:
{
    assert(false);
}
}

switch(var)
{
// for readability, this is also good
case 0:  do_some_thing_short(); break;
case 1:  another_thing_short(); break;
default: assert(false);
}
```

Tip: The space around the operator in loop condition is optional and feel free to insert extra parentheses judi-

ciously for readability.

Pointer Expressions

- No spaces around period or arrow.
- Pointer operators do not have trailing spaces.
- Pointer operators have no space after the * or &.

Examples of correctly-formatted pointer:

```
int  x = *p;
int *z = &x;
int  z = g.y;
int  h = r->y;
```

- When declaring a pointer variable or argument, place the asterisk adjacent to the variable name.

```
char *c;   // Good - variable name just following *, no spaces between them.

char * c;  // Bad  - spaces on both sides of *.
char* c;   // Bad  - space between * and the variable name.
```

- It is not allowed to declare multiple variables in the same declaration.

```
int x, y;  // Bad  - no multiple variables on a declaration.
int a, *b; // Bad  - such declarations are easily misread.

int  x = 2;    // Good - only one variable on a declaration.
int  y = 0;    // Good - easily initiallize it, no misreading.
int  a = 1;
int *b = NULL; // Good - such declaration clear enough.
```

- It is a bad idea to have multiple sentences on the same line.

```
// Bad  - why do you want to do like this?
int x=foo(); char c = get_char();
int a=1; char *str="good";

// Good - why do you make it clear?
int  x = foo();
char c = get_char();
int  a = 1;
char *str = "good";
```

Boolean Expressions

When a boolean expression that is longer than the standard *line length*, break it up by:

- keep operators at the end of the line, and align them for readability and emphasis.
- make all items indent to the first item of the boolean expression.

```
// use minimal lines
if(this_one_thing > this_other_thing && a_third_thing == a_fourth_thing &&
   yet_another_thing && the_last_thing)
{
    // 'yet_another_thing' align to 'this_one_thing'
    ...
}
```

```
// each on a single line, make the operator indented
if(this_one_thing > this_other_thing &&
   a_third_thing == a_fourth_thing   &&
   yet_another_thing                 &&
   the_last_thing)
{
    // all items align to 'this_one_thing'
    ...
}
```

Note:

- Be consistent in how breaking up the lines with the codes around.
- Feel free to insert extra parentheses judiciously because they can be very helpful in increasing readability when used appropriately.
- Always use the punctuation operators, such as `&&` and `~`, rather than the word operators, such as `and` and `compl`.

Return Values

- Do not needlessly surround the return expression with parentheses.
- Use parentheses in return **expr** only where you would use them in `x = expr;` like format.

```
return result;                // Good - No parentheses in the simple case.
return (ret == true);         // Good - return boolean value.
return (sec : opt_1 ? opt_2); // Good - select one as the return value.

// Good - Parentheses OK to make a complex expression more readable.
return (some_long_condition && another_condition);
return (some_long_condition &&
        another_condition   &&
        yes_the_last_one);

return (value);    // Bad - You would never write 'var = (value);', would you ?
return(result);    // Bad - return is not a function!
```

Tip:

- Feel free to insert extra parentheses judiciously because they can be very helpful in increasing readability when used appropriately.

Preprocessor Directives

- The hash mark that starts a preprocessor directive should always be at the beginning of the line.
- Nested directives should add 3 spaces after the hash mark for each level of indentation.
- If preprocessor directives are within the body of indented code, make judiciously indent to increase the readability.

```
if(lopsided_score)
{
#if DISASTER_PENDING
    drop_every_thing();
    // judiciously indent, more readable
    #if NOTIFY
```

```
    notify_client();
    #endif
#endif
    BackToNormal();
}

#ifdef DEBUG_LOG_ENABLE
#   define DEBUG_MSG(msg) printf("%s\n", (msg)); // add 3 spaces before 'define'
#else
#   define DEBUG_MSG(msg)                        // make it more readable
#endif
```

General Horizontal Whitespace

- Use of horizontal whitespace depends on location.
- Never put trailing whitespace at the end of a line.

```
int i = 0;         // Semicolons usually have no space before them.
int x[] = { 0 };  // Spaces inside braces for braced-init-list on both sides.
```

Note: Some editors with `smart` indentation will insert whitespace at the beginning of new lines as appropriate, so you can start typing the next line of code right away. However, if some such editors do not remove the whitespace when you end up not putting a line of code there, such as if you leave a blank line. As a result, you end up with lines containing trailing whitespace.

Warning: Adding trailing whitespace can cause extra work for others editing the same file when they merge, as they can removing existing trailing whitespace, they are invisiable, are't they.
Thus, do NOT introduce trailing whitespace. Remove it if you're already changing that line, or do it in a separate clean-up operation(preferably when no-one else is working on the file.

Blocks Horizontal Whitespace

```
// no space after the keyword in conditions and loops
if(b)
{
    ...
    do_some_thing(); // 4 space indent
}

// usually no space inside parentheses
// no space after the keywords: while
while(test) { continue; }

// no space after the keywords: for
// for loops always have a space after the semicolon
// for loops usually no space before the semicolon
for(int i = 0; i < 5; ++i)
{
    // one space before the semicolon
    for( ; ; )
    {
        ...
        if(condition) break; // 4 space indent
    }
}
```

```
// no space after the keywords: switch
switch(i)
{
case 1:  // No space before colon in a switch case.
{ ... }
case 2:
{ ... }
default: // Always have default
{ ... }
}

// the same goes for union and enum
struct my_struct_ST
{   // open curly brace on the next line by itself
    // 4 space indent
    const char *m_name;  ///< name of people, max len is 100
    const char *m_addr;  ///< home address, max len is 512
    // make properly align of members
    // make properly align of members comments if have
    bool  m_boy;         ///< boy: @b true; girl: @b false
    int   m_age;         ///< age, [1, 150]
};// no space between close curly brace and semicolon

// the same goes for union and enum
typedef struct
{
    const char *m_name;  ///< name of people, max len is 100
    const char *m_addr;  ///< home address, max len is 512
    bool  m_boy;         ///< boy: @b true; girl: @b false
    int   m_age;         ///< age, [1, 150]
} my_struct_st;
// no space between the name and semicolon
// one space between close curly brace and the name
```

Operators Horizontal Whitespace

```
x = 0;              // assignment operators always have spaces around them.
v = w * x + y / z; // binary operators usually have spaces around them.
v = w*x + y/z;      // it's OK to remove spaces around factors, if still clear enough.
v = w * (x + z);    // parentheses should have no internal padding.

// no spaces separating unary operators and their arguments.
x = -5;
++x;
if(x && !y)
{
    ...
}
```

Tip:

- Feel free to insert extra parentheses judiciously because they can be very helpful in increasing readability when used appropriately.

Variables Horizontal Whitespace

```
int long_variable = 0;  // NEVER align assignments like this.
int i             = 1;

int i = 1;              // this will be clear and good enough.
int a_var = 0;
int an_var = 1;
int yes_anox = 5;
int long_variable = 0;

struct my_struct_ST
{
    const char *m_name;
    const char *m_addr;  // make properly align of members
    bool  m_boy;         // make properly align of members
    int   m_age;
} my_variable[] =        // one space between close curly brace and variable
{   // open curly brace on the next line by itself
    // make properly align, increasing the readability
    // make sure no space before the comma
    // 4 space indent
    { "Mia",       "Address",        true,  8  },
    { "Elizabeth", "AnotherAddress", false, 10 },
};
```

Macros Horizontal Whitespace

```
// Align \'s in macro definitions like this, increasing readability
#define __KHASH_TYPE(name, khkey_st, khval_st) \
    typedef struct                             \
    {                                          \
        khint_st    m_buckets;  /* comments */ \
        khint_st    m_size;     /* comments */ \
        khint_st    m_occupied;                \
        khint_st    m_upper_bound;             \
        khint32_st *m_flags;    /* comments */ \
        khkey_st   *m_keys;                    \
        khval_st   *m_vals;                    \
        } kh_##name##_st;

// VS.

#define __KHASH_TYPE(name, khkey_st, khval_st) \
    typedef struct \
    { \
        khint_st    m_buckets; \
        khint_st    m_size; \
        khint_st    m_occupied; \
        khint_st    m_upper_bound; \
        khint32_st *m_flags; \
        khkey_st   *m_keys; \
        khval_st   *m_vals; \
        } kh_##name##_st;

// for readability, this is also make sense
#define A_MACRO         something
#define ANOTHER_MACRO   another_thing
#define YET_ALSO_MACRO  yet_also_something

// if it is to long to fit one line, breaking up like this
```

```
#define A_VERY_LONG_MACRO_NAME \
    a_good_idea_to_have_this_macro_so_long
```

Tip: Feel free to insert extra parentheses or braces judiciously

- Maybe it is necessarily to make sure the code work correctly
- Maybe it will very helpful in increasing readability

Warning: If you can avoid using macros, just do not use them.

Vertical Whitespace

- Minimize use of vertical whitespace.
- Do not end functions with blank lines.
- Do not start functions with blank lines.
- Do not use blank lines when you do not have to.
- Do not put more than one or two blank lines between functions.
- Blank lines inside a chain of if-else blocks may well help readability.
- Blank lines at the beginning or end of a function very rarely help readability.

Tip: The more code that fits on one screen, the easier it is to follow and understand the control flow of the program. Of course, readability can suffer from code being too dense as well as too spread out, so use your judgment. But in general, minimize use of vertical whitespace.

Exceptions

You may diverge from the rules when dealing with code that does not conform to this style guide.

If you find yourself modifying code that was written to specifications other than those presented by this guide, you may have to diverge from these rules in order to stay consistent with the local conventions in that code. If you are in doubt about how to do this, ask the original author or the person currently responsible for the code.

Remember that consistency includes local consistency, too.

Ending

- Use common sense and BE CONSISTENT.
- Take a few minutes to look at the code around you and determine its style.

Enough writing about writing code. Have fun, enjoy coding!

- [1] Google Coding Style

Tip: The style guide is supposed to make the code more readable. If you think you have to violate its rules for the sake of clarity, just do it!

Lua

Types

- **Primitives**: When you access a primitive type you work directly on its value
  - `string`
  - `number`
  - `boolean`
  - `nil`

```
local foo = 1
local bar = foo

bar = 9

print(foo, bar) -- => 1      9
```

- **Complex**: When you access a complex type you work on a reference to its value
  - `table`
  - `function`
  - `userdata`

```
local foo = { 1, 2 }
local bar = foo

bar[0] = 9
foo[1] = 3

print(foo[0], bar[0]) -- => 9    9
print(foo[1], bar[1]) -- => 3    3
print(foo[2], bar[2]) -- => 2    2
```

Tables

- Use the constructor syntax for table property creation where possible.

```
-- bad
local player = {}
player.name  = 'Jack'
player.class = 'Rogue'

-- good
```

```
local player =
{
    name = 'Jack',
    class = 'Rogue'
}
```

- Define functions externally to table definition.

```
-- bad
local player =
{
    attack = function()
    -- ...stuff...
    end
}

-- good
local function attack()
end

local player =
{
    attack = attack
}
```

- Consider `nil` properties when selecting lengths. A good idea is to store an `n` property on lists that contain the length (as noted in Storing Nils in Tables)

```
-- nils don't count
local list = {}
list[0] = nil
list[1] = 'item'

print(#list) -- 0
print(select('#', list)) -- 1
```

- When tables have functions, use `self` when referring to itself.

```
-- bad
local me =
{
    fullname = function(this)
        return this.first_name + ' ' + this.last_name
    end
}

-- good
local me =
{
    fullname = function(self)
        return self.first_name + ' ' + self.last_name
    end
}
```

Strings

- Use single quotes `''` for strings.

```
-- bad
local name = "Bob Parr"
```

```
-- good
local name = 'Bob Parr'

-- bad
local fullName = "Bob " .. self.lastName

-- good
local fullName = 'Bob ' .. self.lastName
```

- Strings longer than 100 characters should be written across multiple lines using **concatenation**.

```
-- good, more readable
local errorMessage = 'This is a super long '        ..
                     'error message line. This is ' ..
                     'a super long error message line.'

-- good, more readable
local errorMessage =
    'This is a super long '        ..
    'error message line. This is ' ..
    'a super long error message line.'

-- bad
local errorMessage = [[This is a super long error that
    was thrown because of Batman.
    When you stop to think about
    how Batman had anything to do
    with this, you would get nowhere
    fast.]]
```

Functions

- Prefer lots of **small** functions to large, complex functions. Smalls Functions Are Good.
- Prefer function syntax over variable syntax. This helps differentiate between named and anonymous functions.

```
-- bad
local nope = function(name, options)
    -- ...stuff...
end

-- good
local function yup(name, options)
    -- ...stuff...
end
```

- Never name a parameter `arg`, this will take precendence over the `arg` object that is given to every function scope in older versions of Lua.

```
-- bad
local function nope(name, options, arg)
    -- ...stuff...
end

-- good
local function yup(name, options, ...)
    -- ...stuff...
end
```

- Perform validation early and return as early as possible.

```
-- bad
local is_good_name = function(name, options, arg)
    local is_good = #name > 3
    is_good = is_good and #name < 30

    -- ...stuff...

    return is_bad
end

-- good
local is_good_name = function(name, options, args)
    if #name < 3 or #name > 30 then return false end

    -- ...stuff...

    return true
end
```

Properties

- Use dot notation when accessing known properties.

```
local luke =
{
    jedi = true,
    age  = 28
}

-- bad
local isJedi = luke['jedi']

-- good
local isJedi = luke.jedi
```

- Use subscript notation `[]` when accessing properties with a variable or if using a table as a list.

```
local luke =
{
    jedi = true,
    age  = 28
}

local function getProp(prop)
    return luke[prop]
end

local isJedi = getProp('jedi')
```

Variables

- Always use `local` to declare variables. Not doing so will result in global variables to avoid polluting the global namespace.

```
-- bad
superPower = SuperPower()

-- good
local superPower = SuperPower()
```

- Assign variables at the top of their scope where possible. This makes it easier to check for existing variables.

```
-- bad
local bad = function()
    test()
    print('doing stuff..')

    //..other stuff..

    local name = getName()

    if name == 'test' then
        return false
    end

    return name
end

-- good
local function good()
    local name = getName()

    test()
    print('doing stuff..')

    //..other stuff..

    if name == 'test' then
        return false
    end

    return name
end
```

Conditional Expressions & Equality

- `false` and `nil` are **falsy** in conditional expressions. All else is **true**.

```
local str = ''

if str then
    -- true
end
```

- Use shortcuts when you can, unless you need to know the difference between `false` and `nil`.

```
-- bad
if name ~= nil then
    -- ...stuff...
end

-- good
if name then
    -- ...stuff...
end
```

- Prefer **true** statements over **false** statements where it makes sense. Prioritize truthy conditions when writing multiple conditions.

```
--bad
if not thing then
    -- ...stuff...
```

```
else
    -- ...stuff...
end

--good
if thing then
    -- ...stuff...
else
    -- ...stuff...
end
```

- Prefer defaults to `else` statements where it makes sense. This results in less complex and safer code at the expense of variable reassignment, so situations may differ.

```
--bad
local function full_name(first, last)
    local name

    if first and last then
        name = first .. ' ' .. last
    else
        name = 'John Smith'
    end

    return name
end

--good
local function full_name(first, last)
    local name = 'John Smith'

    if first and last then
        name = first .. ' ' .. last
    end

    return name
end
```

- Short ternaries are okay.

```
local function default_name(name)
    -- return the default 'Waldo' if name is nil
    return name or 'Waldo'
end

local function brew_coffee(machine)
    return machine and machine.is_loaded and 'coffee brewing' or 'fill your water'
end
```

Blocks

- Single line blocks are okay for **small** statements.
- Try to keep lines length to 100 characters.
- Indent lines if they overflow past the limit.
- Putting multiple statements on one line is discouraged, unless the expression is very short.

```
-- good
if test then return false end
```

```
-- good
if test then
    return false
end

-- bad, the line too long
if test < 1 and do_complicated_function(test) == false or seven == 8 and nine == 10 then do_other_

-- good
if test < 1 and do_complicated_function(test) == false or
   seven == 8 and nine == 10 then

    do_other_complicated_function()
    return false
end
```

Whitespace

- Tabs are 4 characters, and thus indentations are also 4 characters, and use spaces only.

```
-- bad  - 1 space indentations
function()
 local name
end

-- bad  - 2 space indentations
function()
  local name
end

-- bad  - 3 space indentations
function()
   local name
end

-- Good - 4 space indentations
function()
    local name
end
```

- Place 1 space before opening and closing braces.
- Place no spaces around parens.

```
-- bad
local test = {one=1}

-- good
local test = { one = 1 }

-- bad
dog.set('attr',{
    age = '1 year',
    breed = 'Bernese Mountain Dog'
})

-- good
dog.set('attr', {
    age = '1 year',
    breed = 'Bernese Mountain Dog'
})
```

- Surround operators with spaces.

```
-- bad
local thing=1
thing = thing-1
thing = thing*1
thing = 'string'..'s'

-- good
local thing = 1
thing = thing - 1
thing = thing * 1
thing = 'string' .. 's'
```

- Use one space after commas.

```
--bad
local thing = {1,2,3}
thing = {1 , 2 , 3}
thing = {1 ,2 ,3}

--good
local thing = {1, 2, 3}
```

- Add a line break after multiline blocks.

```
--bad
if thing then
    -- ...stuff...
end
function derp()
    -- ...stuff...
end
local wat = 7

--good
if thing then
    -- ...stuff...
end

function derp()
    -- ...stuff...
end

local wat = 7
```

- Place an empty newline at the end of the file.
- Delete unnecessary whitespace at the end of lines.
- No spaces should be used immediately before or inside a bracketing character.
  - `[`, `(`, `{` and their matches.

Commas

- Leading commas aren't okay. An ending comma on the last item is okay but discouraged.

```
-- bad, why do you want write it like this?
local thing =
{
    once = 1
    , upon = 2
```

```
    , aTime = 3
}

-- good
local thing =
{
    once = 1,
    upon = 2,
    aTime = 3
}

-- okay
local thing =
{
    once = 1,
    upon = 2,
    aTime = 3,
}
```

Semicolons

- Separate statements onto multiple lines.

```
-- bad
local whatever = 'sure';
a = 1; b = 2

-- good and readable
local whatever = 'sure'
a = 1
b = 2
```

Type Casting & Coercion

- Perform type coercion at the beginning of the statement. Use the built-in functions.
  - built-in functions: `tostring`, `tonumber`, etc.
- Use `tostring` for strings if you need to cast without string concatenation.

```
-- bad
local totalScore = reviewScore .. ''

-- good
local totalScore = tostring(reviewScore)
```

- Use `tonumber` for Numbers.

```
local inputValue = '4'

-- bad
local val = inputValue * 1

-- good
local val = tonumber(inputValue)
```

Naming Conventions

- Avoid single letter names.
- Be descriptive with your naming.
- You can get away with single-letter names when they are variables in loops.

```
-- bad
local function q()
    -- ...stuff...
end

-- good
local function query()
    -- ..stuff..
end
```

- Use underscores for ignored variables in loops.

```
--good
for _, name in pairs(names) do
    -- ...stuff...
end
```

- Use **snake case** when naming objects, functions, and instances. Tend towards verbosity if unsure about naming.

```
-- bad
local OBJEcttssss = {}
local thisIsMyObject = {}
local this-is-my-object = {}

local c = function()
    -- ...stuff...
end

-- good
local this_is_my_object = {}

local function do_that_thing()
    -- ...stuff...
end
```

- Use **PascalCase** for factories.

```
-- bad
local player = require('player')

-- good
local Player = require('player')
local me = Player({ name = 'Jack' })
```

- Use **is** or **has** prefix for boolean-returning functions that are part of tables.

```
--bad
local function evil(alignment)
    return alignment < 100
end

--good
local function is_evil(alignment)
    return alignment < 100
end
```

Constructors

Modules

- The module should return a table or function.
- The module should not use the global namespace for anything ever.
- The module should be a closure.
- The file should be named like the module.

```
-- thing.lua
local thing = { }

local meta =
{
    __call = function(self, key, vars)
    print key
    end
}


return setmetatable(thing, meta)
```

- Note that modules are loaded as singletons and therefore should usually be factories (a function returning a new instance of a table) unless static (like utility libraries.)

File Structure

- Files should be named in all lowercase.
- Lua files should be in a top-level **src** folder.
- The main library file should be called `module_name.lua`.
- **LICENSE**, **README**, etc should be in the top level.
- Tests should be in a top-level **test** folder.
- Executables should be in a top-level **bin** folder.

```
-- File Structure of lua project or module
--
-- ./a_lua_module
--      bin/
--          script.sh

--      test/
--          test_module.lua
--          test_some_file.lua

--      src/
--          module.lua
--          some_file.lua

--      README.md
--      LICENSE.md
```

Testing

- Use busted and write lots of tests in a `test` folder. Separate tests by module.
- Use descriptive `describe` and `it` blocks so it's obvious to see what precisely is failing.
- Test interfaces. Don't test private methods. If you need to test something that is private, it probably shouldn't be private in the first place.

Resources

Ending

- Use **common** sense and be **consistency**.
- Take a few minutes to look at the code around you and determine its style.

Enough writing about writing code. Have fun, enjoy coding!

- [1] Lua Style Guide
- [2] MediaWiki Lua Coding Conventions

Ending

- Use **common** sense and be **consistency**.
- Take a few minutes to look at the code around you and determine its style.

Enough writing about writing code. Have fun, enjoy coding!

- [1] Google XML Style Guide

JSON

Ending

- Use **common** sense and be **consistency**.
- Take a few minutes to look at the code around you and determine its style.

Enough writing about writing code. Have fun, enjoy coding!

- [1] Google JSON Style Guide

CHAPTER 6

Shell

Background

Which Shell to Use

- `Bash` is the only shell scripting language permitted for executables
- Executables must start with `!/usr/bin/env bash` shebang and a minimum number of flags
  - Bash is not always located at `/bin/bash`, do it like this make sense.
- Use `set` to set shell options
  - do that make calling your script as `bash script_name` does not break its functionality
- memorize and utilize `set -eu -o pipefail` at the very beginning
  - never write a script without `set -e` at the very very beginning.

    This instructs bash to terminate in case a command or chain of command finishes with a non-zero exit status. The idea behind this is that a proper programm should never have unhandled error conditions. Use constructs like `if myprogramm --parameter ; then ...` for calls that might fail and require specific error handling. Use a cleanup trap for everything else.
  - use `set -u` in your scripts.

  This will terminate your scripts in case an uninitialized variable is accessed. This is especially important when developing shell libraries, since library code accessing uninitialized variables will fail in case it's used in another script which sets the `-u` flag. Obviously this flag is relevant to the script's/code's security.
  - use `set -o pipefail` to get an exit status from a pipeline (last non-zero will be returned).

Restricting all executable shell scripts to `bash` gives us a consistent shell language that's installed on all our machines.

The only exception to this is where you're forced to by whatever you're coding for.

When to use Shell

Shell should only be used for small utilities or simple wrapper scripts.

While shell scripting isn't a development language, it is used for writing various utility scripts. This style guide is more a recognition of its use rather than a suggestion that it be used for widespread deployment.

- If you're mostly calling other utilities and are doing relatively little data manipulation, shell is an acceptable choice for the task.
- If performance matters, use something other than shell.

- If you find you need to use arrays for anything more than assignment of `${PIPESTATUS}`, you should use Python.
- If you are writing a script that is more than 100 lines long, you should probably be writing it in Python instead. Bear in mind that scripts grow. Rewrite your script in another language early to avoid a time-consuming rewrite at a later date.

File Extensions

Executables should have no extension (strongly preferred) or a `.sh` extension. Libraries must have a `.sh` extension and should not be executable.

It is not necessary to know what language a program is written in when executing it and shell does not require an extension so we prefer not to use one for executables.

However, for libraries it's important to know what language it is and sometimes there's a need to have similar libraries in different languages. This allows library files with identical purposes but different languages to be identically named except for the language-specific suffix.

SUID and SGID

SUID and SGID are **forbidden** on shell scripts.

There are too many security issues with shell that make it nearly impossible to secure sufficiently to allow SUID/SGID. While bash does make it difficult to run SUID, it's still possible on some platforms which is why we're being explicit about banning it.

Use `sudo` to provide elevated access if you need it.

Environment

stdout VS stderr

All error messages should go to `stderr`. This makes it easier to separate normal status from actual issues.

A function to print out error messages along with other status information is recommended, e.g.

```
err()
{
    echo "[$(date +'%Y-%m-%dT%H:%M:%S%z')]: $@" >&2
}

if ! do_something; then
    err "Unable to do_something"
    exit "${E_DID_NOTHING}"
fi
```

Comments

File Header

Start each file with a description of its contents.

Every file must have a top-level comment including a brief overview of its contents. A copyright notice and author information are optional, for example:

```
#!/bin/bash
#
# Perform hot backups of Oracle databases.
...
the_body_of_scripts
...
```

Function Comments

- Any function that is not both obvious and short must be commented.
- Any function in a library must be commented regardless of length or complexity.

It should be possible for someone else to learn how to use your program or to use a function in your library by reading the comments (and self-help, if provided) without reading the code. All function comments should contain:

- Description of the function
- Global variables used and modified
- Arguments taken
- Returned values other than the default exit status of the last command run

For example:

```
#!/bin/bash
#
# File description information

export PATH='/usr/xpg4/bin:/usr/bin:/opt/csw/bin:/opt/goog/bin'

########################################
# Cleanup files from the backup dir
# Globals:
#     BACKUP_DIR
#     ORACLE_SID
# Arguments:
#     None
# Returns:
#     None
########################################
cleanup()
{
    ...
    function_body
    ...
}
```

Implementation Comments

Comment tricky, non-obvious, interesting or important parts of your code.

This follows general Google coding comment practice. Don't comment everything. If there's a complex algorithm or you're doing something out of the ordinary, put a short comment in.

TODO Comments

Use TODO comments for code that is temporary, a short-term solution, or good-enough but not perfect.

TODOs should include the string TODO in all caps, followed by your username in parentheses. A colon is optional. It's preferable to put a bug/ticket number next to the TODO item as well.

For examples:

```
# TODO(mrmonkey): Handle the unlikely edge cases (bug ####)
```

Formatting

Indentation

Indent 4 spaces for each level. No tabs and only spaces.

- Use blank lines between blocks to improve readability.
- Indentation is 4 spaces for each level.
- Whatever you do, don't use tabs, and always use spaces.

Line Length and Long Strings

Maximum line length is 100 characters.

If you have to write strings that are longer than 100 characters, this should be done with a here document or an embedded newline if possible. Literal strings that have to be longer than 100 chars and can't sensibly be split are ok, but it's strongly preferred to find a way to make it shorter. e.g.

```
# DO use 'here documention'
cat <<END
I am an exceptionally long
string.
END

# Embedded newlines are ok too
long_string="I am an exceptionally
             long string."
```

Pipelines

Pipelines should be split one per line if they don't all fit on one line.

If a pipeline all fits on one line, it should be on one line. If not, it should be split at one pipe segment per line with the pipe on the end of the line, and align the new line to the first item. This applies to a chain of commands combined using `|` as well as to logical compounds using `||` and `&&`, e.g.

```
# All fits on one line of pipline
ls ${long_list_of_parameters} | grep ${foo} | grep -v grep | pgrep | wc -l | sort | uniq

# Long commands of pipline, far more readable, isn't it?
ls ${long_list_of_parameters}         \
   grep ${foo}                    | \
   grep -v grep                   | \
   pgrep                          | \
   wc -l                          | \
   sort                           | \
   uniq
```

Loops and Conditions

Put `; do` and `; then` on the same line as the `while`, `for` or `if`.

Loops in shell are a bit different, but we follow the same principles as with braces when declaring functions. That is: `; then` and `; do` should be on the same line as the `if`, `for` and `while`. while `else` should be on its own line and closing statements should be on their own line vertically aligned with the opening statement, For example:

```
if ${event}; then
    ...
fi

while ${event}; do
    ...
done

for v in ${list[@]}; do
    ...
done

for dir in ${dirs_to_cleanup}; do
    if [[ -d "${dir}/${ORACLE_SID}" ]]; then
        log_date "Cleaning up old files in ${dir}/${ORACLE_SID}"
        rm "${dir}/${ORACLE_SID}/"*
        if [[ "$?" -ne 0 ]]; then
            error_message
        fi
    else
        mkdir -p "${dir}/${ORACLE_SID}"
        if [[ "$?" -ne 0 ]]; then
            error_message
        fi
    fi
done
```

Case Statement

- Indent by 4 spaces.
- A one-line needs a space after the close parenthesis of the pattern and before the `;;`
- Long or multi-command should be split over multiple lines with the pattern, actions, and `;;` on separate lines.

The matching expressions are indented one level from the `case` and `esac`. Multiline actions are indented another level. In general, there is no need to quote match expressions. Pattern expressions should not be preceded by an open parenthesis. Avoid the `&` and `;;&` notations.

For example:

```
case "${expression}"; in
    a)
        variable="..."
        some_command "${variable}" "${other_expr}" ...
        ;;
    absolute)
        actions="relative"
        another_command "${actions}" "${other_expr}" ...
        ;;
    *)
        error "Unexpected expression '${expression}'"
```

```
    ;;
esac
```

Simple commands may be put on the same line as the pattern and `;;` as long as the expression remains readable. This is often appropriate for single-letter option processing. When the actions don't fit on a single line, put the pattern on a line on its own, then the actions, then `;;` also on a line of its own. When on the same line as the actions, use a space after the close parenthesis of the pattern and another before the `;;`.

```
verbose='false'
aflag=''
bflag=''
files=''
while getopts 'abf:v' flag; do
    case "${flag}" in
        a) aflag='true' ;;
        b) bflag='true' ;;
        f) files="${OPTARG}" ;;
        v) verbose='true' ;;
        *) error "Unexpected option ${flag}" ;;
    esac
done
```

Variable expansion

In order of precedence:

- stay consistent with what you find, quote your variables
- prefer `${var}` over `$var`

These are meant to be guidelines, as the topic seems too controversial for a mandatory regulation.

They are listed in order of precedence.

- Stay consistent with what you find for existing code
- *Quote variables*
- Don't brace-quote single character shell **specials/positional** parameters, unless strictly necessary or avoiding deep confusion. Prefer brace-quoting all other variables

```
# Preferred style for 'special' variables:
echo "Positional: $1" "$5" "$3"
echo "Specials: !=$!, -=$-, _=$_. ?=$?, #=$# *=$* @=$@ \$=$$ ..."

# Braces necessary:
echo "many parameters: ${10}"

# Braces avoiding confusion:
# Output is "a0b0c0"
set -- a b c
echo "${1}0${2}0${3}0"

# Preferred style for other variables:
echo "PATH=${PATH}, PWD=${PWD}, mine=${some_var}"
while read f; do
    echo "file=${f}"
done < <(ls -l /tmp)

# Section of discouraged cases

# Unquoted vars, unbraced vars, brace-quoted single letter
# shell specials.
echo a=$avar "b=$bvar" "PID=${$}" "${1}"
```

```
# Confusing use: this is expanded as "${1}0${2}0${3}0",
# not "${10}${20}${30}
set -- a b c
echo "$10$20$30"
```

Quoting

- Use double quotes for strings that require variable expansion or command substitution interpolation and single quotes for all others.
- Be aware of the quoting rules for pattern matches.
  - Single quotes indicate that no substitution is desired.
  - Double quotes indicate that substitution is required/tolerated.

```
# right
foo='Hello World'
bar="You are $USER"

# wrong
foo="hello world"

# possibly wrong, depending on intent
bar='You are $USER'
```

- Always quote strings containing variables, command substitutions, spaces or shell meta characters, unless careful unquoted expansion is required.
- Never quote **literal** integers.
- Use `$@` unless you have a specific reason to use `$*`.

```
# Simple examples
#
# quote command substitutions
flag="$(some_command and its args "$@" 'quoted separately')"
#
# quote variables
echo "${flag}"
#
# never quote literal integers
value=32
#
# quote command substitutions, even when you expect integers
number="$(generate_number)"
#
# prefer quoting words, not compulsory
readonly USE_INTEGER='true'
#
# quote shell meta characters
echo 'Hello stranger, and well met. Earn lots of $$$'
echo "Process $$: Done making \$\$\$."
#
# command options or path names
# $1 is assumed to contain a value here
grep -li Hugo /dev/null "$1"
#
# Less simple examples
# quote variables, unless proven false: ccs might be empty
git send-email --to "${reviewers}" ${ccs:+"--cc" "${ccs}"}
#
# Positional parameter precautions: $1 might be unset
```

```
# Single quotes leave regex as-is.
grep -cP '([Ss]pecial|\|?characters*)$' ${1:+"$1"}
#
# For passing on arguments,
# "$@" is right almost everytime, and $* is wrong almost everytime:
#
# > $* and $@ will split on spaces, clobbering up arguments
#   that contain spaces and dropping empty strings;
# > "$@" will retain arguments as-is, so no args
#   provided will result in no args being passed on;
#   This is in most cases what you want to use for passing
#   on arguments.
# > "$*" expands to one argument, with all args joined
#   by (usually) spaces,
#   so no args provided will result in one empty string
#   being passed on.
#
#   Consult 'man bash' for the nit-grits ;-)
#
set -- 1 "2 two" "3 three tres"; echo $# ; set -- "$*"; echo "$#, $@")
set -- 1 "2 two" "3 three tres"; echo $# ; set -- "$@"; echo "$#, $@")
```

When in doubt however, quote all expansions.

Shell Features

Command Substitution

Use `$(command)` instead of backticks(`` `command` ``).

- Nested backticks require escaping the inner ones with \
- The `$(command)` format doesn't change when nested and is easier to read

```
# This is preferred:
var="$(command "$(command1)")"

# This is not:
var="`command \`command1\``"
```

Math and Integer Manipulation

Use `((...))` and `$((...))`.

```
a=5
b=4

# wrong
if [[ $a -gt $b ]]; then
    ...
fi

# right
if ((a > b)); then
    ...
fi
```

Test, [and [[

`[[ ... ]]` is preferred over `[`, `test` and `/usr/bin/[`, see BashFAQ for more information.

- `[[ ... ]]` reduces errors as no pathname expansion or word splitting takes place
- `[[ ... ]]` allows for regular expression matching where `[ ... ]` does not

```
# This ensures the string on the left is made up of characters in the
# alnum character class followed by the string name.
# Note that the RHS should not be quoted here.
# For the gory details, see
# E14 at https://tiswww.case.edu/php/chet/bash/FAQ
if [[ "filename" =~ ^[[:alnum:]]+name ]]; then
    echo "Match"
fi

# This matches the exact pattern "f*" (Does not match in this case)
if [[ "filename" == "f*" ]]; then
    echo "Match"
fi

# This gives a "too many arguments" error as f* is expanded to the
# contents of the current directory
if [ "filename" == f* ]; then
    echo "Match"
fi
```

Testing Strings

Use quotes rather than filler characters where possible.

Bash is smart enough to deal with an empty string in a test. So, given that the code is much easier to read, use tests for **non-empty** strings or **empty** strings rather than **filler** characters.

```
# Do this:
if [[ "${my_var}" = "some_string" ]]; then
    do_something
fi

# -z (string length is zero) and -n (string length is not zero) are
# preferred over testing for an empty string
if [[ -z "${my_var}" ]]; then
    do_something
fi

# This is OK (ensure quotes on the empty side), but not preferred:
if [[ "${my_var}" = "" ]]; then
    do_something
fi

# Not this:
if [[ "${my_var}X" = "some_stringX" ]]; then
    do_something
fi
```

To avoid confusion about what you're testing for, explicitly use `-z` or `-n`.

```
# Use this
if [[ -n "${my_var}" ]]; then
    do_something
fi
```

```
# Instead of this as errors can occur if ${my_var} expands to a test
# flag
if [[ "${my_var}" ]]; then
    do_something
fi
```

Wildcard Expansion of Filenames

Use an explicit path when doing wildcard expansion of filenames.

As filenames can begin with a −, it's a lot safer to expand wildcards with `./*` instead of `*`.

```
# Here's the contents of the directory:
# -f  -r  somedir  somefile

# This deletes almost everything in the directory by force
psa@bilby$ rm -v *
removed directory: `somedir'
removed `somefile'

# As opposed to:
psa@bilby$ rm -v ./*
removed `./-f'
removed `./-r'
rm: cannot remove `./somedir': Is a directory
removed `./somefile'
```

Eval

`eval` should be avoided.

Eval munges the input when used for assignment to variables and can set variables without making it possible to check what those variables were.

```
# What does this set?
# Did it succeed? In part or whole?
eval $(set_my_variables)

# What happens if one of the returned values has a space in it?
variable="$(eval some_function)"
```

Pipes to While

Use process substitution or for loops in preference to piping to while. Variables modified in a while loop do not propagate to the parent because the loop's commands run in a subshell.

The implicit subshell in a pipe to while can make it difficult to track down bugs.

```
last_line='NULL'
your_command | while read line; do
    last_line="${line}"
done

# This will output 'NULL'
echo "${last_line}"
```

Use a for loop if you are confident that the input will not contain spaces or special characters, usually, this means not user input.

```
total=0
# Only do this if there are no spaces in return values.
for value in $(command); do
    total+="${value}"
done
```

Using process substitution allows redirecting output but puts the commands in an explicit subshell rather than the implicit subshell that bash creates for the while loop.

```
total=0
last_file=
while read count filename; do
    total+="${count}"
    last_file="${filename}"
done < <(your_command | uniq -c)

# This will output the second field of the last line of output from
# the command.
echo "Total = ${total}"
echo "Last one = ${last_file}"
```

Use while loops where it is not necessary to pass complex results to the parent shell. This is typically where some more complex **parsing** is required. Beware that simple examples are probably more easily done with a tool such as awk. This may also be useful where you specifically don't want to change the parent scope variables.

```
# Trivial implementation of awk expression:
#   awk '$3 == "nfs" { print $2 " maps to " $1 }' /proc/mounts
cat /proc/mounts | while read src dest type opts rest; do
    if [[ ${type} == "nfs" ]]; then
        echo "NFS ${dest} maps to ${src}"
    fi
done
```

Read-only Variables

Use `readonly` or `declare -r` to ensure they're read only.

As globals are widely used in shell, it's important to catch errors when working with them. When you declare a variable that is meant to be read-only, make this explicit.

```
zip_version="$(dpkg --status zip | grep Version: | cut -d ' ' -f 2)"
if [[ -z "${zip_version}" ]]; then
    error_message
else
    readonly zip_version
fi
```

Use Local Variables

- Declare function-specific variables with `local`.
- Declaration and assignment should be on different lines.

Ensure that local variables are only seen inside a function and its children by using `local` when declaring them. This avoids polluting the global name space and inadvertently setting variables that may have significance outside the function.

Declaration and assignment must be separate statements when the assignment value is provided by a command substitution; as the `local` builtin does not propagate the exit code from the command substitution.

```
my_func2()
{
    local name="$1"

    # Separate lines for declaration and assignment:
    local my_var
    my_var="$(my_func)" || return

    # DO NOT do this: $? contains the exit code of 'local', not my_func
    local my_var="$(my_func)"
    [[ $? -eq 0 ]] || return

    ...
}
```

Function Location

- Put all functions together in the file just below constants
  - If you've got functions, put them all together near the top of the file
  - Only includes, set statements and setting constants may be done before declaring functions
- Don't hide executable code between functions
  - Don't hide executable code between functions. Doing so makes the code difficult to follow and results in nasty surprises when debugging

main Function

A function called **main** is required for scripts long enough to contain at least one other function.

In order to easily find the start of the program, put the main program in a function called **main** as the bottom most function. This provides consistency with the rest of the code base as well as allowing you to define more variables as `local`, which can't be done if the main code is not a function. The last non-comment line in the file should be a call to **main**:

```
main "$@"
```

Note: Obviously, for short scripts where it's just a linear flow, **main** is overkill and so is not required.

Naming Conventions

Function Names

- Lower-case, with underscores to separate words
  - If writing single functions, use lowercase and separate words with underscore
- Separate libraries with `::`
  - If writing a package, separate package names with `::`
- Parentheses are required after the function name
  - Braces must be on the same line as the function name
  - No space between the function name and the parenthesis

- The keyword `function` is optional, but must be used consistently throughout a project.

```
# Single function
my_func()
{
    do_some_thing()
    ...
}

# Part of a package
mypackage::my_func()
{
    do_some_thing()
    ...
}
```

Note: The `function` keyword is extraneous when `()` is present after the function name, but enhances quick identification of functions.

Variable Names

As for *function names*.

Variables names for loops should be similarly named for any variable you're looping through.

```
for zone in ${zones}; do
    something_with "${zone}"
done
```

Constants and Environment Variable Names

All caps, separated with underscores, declared at the top of the file.

- Constants and anything exported to the environment should be capitalized.

```
# Constant
readonly PATH_TO_FILES='/some/path'

# Both constant and environment
declare -xr ORACLE_SID='PROD'
```

Some things become constant at their first setting (for example, via `getopts`). Thus, it's OK to set a constant in getopts or based on a condition, but it should be made readonly immediately afterwards. Note that declare doesn't operate on global variables within functions, so readonly or export is recommended instead.

```
VERBOSE='false'
while getopts 'v' flag; do
    case "${flag}" in
        v) VERBOSE='true' ;;
    esac
done
readonly VERBOSE
```

Source File Names

Filenames should be all lowercase with hyphens between words, if necessary can have digital numbers.

Calling Commands

Checking Return Values

- Always check return values and give informative return values.

For unpiped commands, use `$?` or check directly via an `if` statement to keep it simple.

```
if ! mv "${file_list}" "${dest_dir}/" ; then
    echo "Unable to move ${file_list} to ${dest_dir}" >&2
    exit "${E_BAD_MOVE}"
fi

# Or
mv "${file_list}" "${dest_dir}/"
if [[ "$?" -ne 0 ]]; then
    echo "Unable to move ${file_list} to ${dest_dir}" >&2
    exit "${E_BAD_MOVE}"
fi
```

Bash also has the `PIPESTATUS` variable that allows checking of the return code from all parts of a pipe. If it's only necessary to check success or failure of the whole pipe, then the following is acceptable:

```
tar -cf - ./* | ( cd "${dir}" && tar -xf - )
if [[ "${PIPESTATUS[0]}" -ne 0 || "${PIPESTATUS[1]}" -ne 0 ]]; then
    echo "Unable to tar files to ${dir}" >&2
fi
```

However, as `PIPESTATUS` will be overwritten as soon as you do any other command, if you need to act differently on errors based on where it happened in the pipe, you'll need to assign PIPESTATUS to another variable immediately after running the command, don't forget that `[` is a command and will wipe out `PIPESTATUS`.

```
tar -cf - ./* | ( cd "${DIR}" && tar -xf - )
return_codes=(${PIPESTATUS[*]})
if [[ "${return_codes[0]}" -ne 0 ]]; then
    do_something
fi
if [[ "${return_codes[1]}" -ne 0 ]]; then
    do_something_else
fi
```

Builtin Commands vs. External Commands

Given the choice between invoking a shell **builtin** command and invoking a **external** command, choose the **builtin**. We prefer the use of **builtins** as it's more robust and portable, especially when compared to things like `sed`.

```
# Prefer this:
addition=$((${X} + ${Y}))
substitution="${string/#foo/bar}"

# Instead of this:
addition="$(expr ${X} + ${Y})"
substitution="$(echo "${string}" | sed -e 's/^foo/bar/')"
```

Best Practice Tips

Tip-1

Always set local function variables `local`.

Tip-2

Write clear code.

- never obfuscate what the script is trying to do
- never shorten uncessesarily with a lot of commands per LoC chained with a semicolon

Tip-3

Bash does not have a concept of public and private functions, thus;

- public functions get generic names, whereas
- private functions are prepended by two underscores (RedHat convention)

Tip-4

Stick to the `pushd`, `popd`, and `dirs` builtins for directory stack manipulation where sensible.

Tip-5

Use the builtin `readonly` when declaring constants and immutable variable.

Tip-6

Write generic small check functions instead of large init and clean-up code.

```
# both functions return non-zero on error
function is_valid_string?()
{
    [[ $@ =~ ^[A-Za-z0-9]*$ ]]
}

function is_integer?()
{
    [[ $@ =~ ^-?[0-9]+$ ]]
}
```

Tip-7

If a project gets bigger, split it up into smaller files with clear and obvious naming scheme.

Tip-8

Clearly document code parts that are not easily understood.

Tip-9

Try to stick to restricted mode where sensible and possible to use `set -r`.

> **Use with caution**. While this flag is very useful for security sensitive environments, scripts have to be written with the flag in mind. Adding restricted mode to an existing script will most likely break it.

Tip-10

Scripts should somewhat reflect the following general layout, e.g.

```
#!/usr/bin/env bash
#
# AUTHORS, LICENSE and DOCUMENTATION
#
set -eu -o pipefail

Readonly Variables
Global Variables

Import ("source scriptname") of external source code

Functions
  - function local variables
  - clearly describe interfaces: return either a code or string

Main
  - option parsing
  - log file and syslog handling
  - temp. file and named pipe handling
  - signal traps

------------------------------------------------------------------------
To keep in mind:
  - quoting of all variables passed when executing sub-shells or cli tools.
  - testing of functions, conditionals and flow.
  - makes restricted mode ("set -r") for security sense.
```

Tip-11

Silence is golden, like in any UNIX programm, avoid cluttering the terminal with useless output.

Tip-12

Overusing `grep` and `grep -v`, e.g.

```
ps ax | grep ${processname} | grep -v grep
```

versus using appropriate userland utilities:

```
pgrep ${processname}
```

Tip-13

Please use `printf` instead of `echo`.

the bash builtin `printf` should be preferred to `echo` where possible. It does work like printf in C or any other high-level language, also see builtin printf.

Tip-14

Anonymous Functions, which is also called Lambdas.

Yup, it's possible. But you'll probably never need them, in case you really do, here's how:

```
function lambda()
{
    _f=${1};
    shift
    function _l
    {
        eval ${_f};
    }
    _l ${*};
    unset _l
}
```

Tip-15

Never forget that you cannot put a space/blank around = during an assignment.

```
# This is not work
ret = false

# This works fine
          var_1=1
     a_long_var=2
a_very_long_var=3
```

Extra Docs

- Bash Reference Manual(GNU)
- BashWeaknesses
- BashPitfalls
- BashFAQ
- BashHackers
- docopts project, and its home-page

Ending

- Use **common** sense and be **consistency**.
- Take a few minutes to look at the code around you and determine its style.

Enough writing about writing code. Have fun, enjoy coding!

- [1] Google Shell Style Guide
- [2] Community Bash Style Guide
- [3] Bash Style Guide

Python

Ending

- Use **common** sense and be **consistency**.
- Take a few minutes to look at the code around you and determine its style.

Enough writing about writing code. Have fun, enjoy coding!

- [1] Google Python Style Guide
- [2] MediaWiki Python Coding Conventions
- [3] PEP 8 Style Guide for Python Code
- [4] The Hitchhiker's Guide to Python Code Style

CHAPTER 8

Git Style

Branches

- Choose short and descriptive names, and use dashes to separate words:

```
# good and clear enough
$ git checkout -b oauth-migration

# bad - it is not that clear
$ git checkout -b login_fix
```

- Identifiers from corresponding tickets in an external service are also good candidates, e.g.

```
# GitHub issue #15
$ git checkout -b issue-15
```

- When several people are working on the same feature, it might be convenient to have personal feature branches and a team-wide feature branch. Use the following naming convention:

```
$ git checkout -b feature-a/master # team-wide branch
$ git checkout -b feature-a/maria  # Maria's personal branch
$ git checkout -b feature-a/nick   # Nick's personal branch
```

Merge at will the personal branches to the team-wide branch. Eventually, the team-wide branch will be merged to **master**.

- Delete your branch from the upstream repository after it's merged, unless there is a specific reason not to.

```
# Use the following command while being on "master", to list merged branches:
$ git branch --merged | grep -v "\*"
```

Commits

Each commit should be a single logical change.

- Do not make several logical changes in one commit. For example, if a patch fixes a bug and optimizes the performance of a feature, split it into two separate commits.
- Do not split a single logical change into several commits. For example, the implementation of a feature and the corresponding tests should be in the same commit.

  Use `git add -p` to interactively stage specific portions of the modified files.

Commit early and often.

- Small, self-contained commits are easier to understand and revert when something goes wrong.

Commits should be ordered logically.

- For example, if commit X depends on changes done in commit Y, then commit Y should come before commit X.

Note: While working alone on a local branch that has not yet been pushed, it's fine to use commits as temporary snapshots of your work. However, it still holds true that you should apply all of the above before pushing it.

Messages

- Use the editor, not the terminal, when writing a commit message.

```
# good
$ git commit

# bad
$ git commit -m "Quick fix"
```

- The summary line should be descriptive yet succinct.
  - Ideally, it should be no longer than 50 characters.
  - It should be capitalized and written in imperative present tense.

```
# good - imperative present tense, capitalized, fewer than 50 characters
Mark huge records as obsolete when clearing hinting faults
```

- After that should come a blank line followed by a more thorough description.
  - It should be wrapped to 96 characters and explain why the change is needed, how it addresses the issue and what side-effects it might have.
  - It should also provide any pointers to related resources (eg. link to the corresponding issue in a bug tracker).
  - when writing a commit message, think about what you would need to know if you run across the commit in a year from now.

```
Short (50 chars or fewer) summary of changes

More detailed explanatory text, if necessary. Wrap it to 96 characters. In some contexts,
the first line is treated as the subject of an email and the rest of the text as the body.
The blank line separating the summary from the body is critical, unless you omit the body
entirely, tools like rebase can get confused if you run the two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Use a hyphen or an asterisk for the bullet, followed by a single space, with blank lines
  between each one of them.

Source http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html
```

- If a commit A depends on commit B, the dependency should be stated in the message of commit A. Use the SHA1 when referring to commits.

  Similarly, if commit A solves a bug introduced by commit B, it should also be stated in the message of commit A.

- If a commit is going to be squashed to another commit use the `--squash` and `--fixup` flags respectively, in order to make the intention clear:

```
# Use the --autosquash flag when rebasing.
# The marked commits will be squashed automatically.
$ git commit --squash f387cab2
```

Merging

- Do not rewrite published history.

  The repository's history is **valuable** in its own right and it is very important to be able to tell what actually happened. Altering published history is a common source of problems for anyone working on the project. However, there are cases where rewriting history is legitimate. These are when:

  - You are the only one working on the branch and it is not being reviewed.
  - You want to tidy up your branch or rebase it onto the **master** in order to merge it later.

- Keep the history clean and simple.
  - Make sure it conforms to the style guide and perform any needed actions if it doesn't
  - Rebase it onto the branch it's going to be merged to

```
[my-branch] $ git fetch
[my-branch] $ git rebase origin/master
# then merge
```

- If your branch includes more than one commit, do not merge with a fast-forward.

```
# good - ensures that a merge commit is created
$ git merge --no-ff my-branch

# bad
$ git merge my-branch
```

Misc

- There are various workflows and each one has its strengths and weaknesses.
- Test before you push. Do not push half-done work.
- Use annotated tags for marking releases or other important points in the history.
- Prefer lightweight tags for personal use, such as to bookmark commits for future reference.
- Keep your repositories at a good shape by performing maintenance tasks occasionally.
  - git-gc Cleanup unnecessary files and optimize the local repository
  - git-prune Prune all unreachable objects from the object database
  - git-fsck Verifies the connectivity and validity of the objects in the database

Tip: Be consistent. This is related to the workflow but also expands to things like commit messages, branch names and tags. Having a consistent style throughout the repository makes it easy to understand what is going on by looking at the log, a commit message etc.

Ending

- [1] Git Style Guide
- [2] Kernel Git Style
- [3] Git Reference

CHAPTER 9

Appendix

Change Log

What is a change log?

A change log is a file which contains a curated, chronologically ordered list of notable changes for each version of a project.

What is the point of a change log?

To make it easier for users and contributors to see precisely what notable changes have been made between each version of the project.

Why should I care?

Because software tools are for people. If you donâĂŹt care, why are you contributing to open source? Surely, there must be a kernel of care somewhere in that lovely little brain of yours.

What should the change-log file be named?

- `LICENSE` is the best convention so far.
- `README.md` is the best convention so far.
- `CHANGELOG.md` is the best convention so far.
- `CONTRIBUTING.md` is the best convention so far.

What makes a good change log?

A good CHANGELOG.md file sticks to the following principles:

- It's made for humans, not machines, so legibility is crucial.
- Easy to link to any section, hence Markdown over plain text.
- One sub-section per version.
- List releases in reverse-chronological order, which is newest on top.
- Write all dates in YYYY-MM-DD format, e.g. `2012-06-02` for `June 2nd, 2012`.
- Explicitly mention whether the project follows Semantic Versioning.

For each version in change log should following:

- List its release date in `YYYY-MM-DD` format.
- Group changes to describe their impact on the project, as follows:
  - **[Fixed]** for any bug fixes.
  - **[Added]** for new features.
  - **[Removed]** for deprecated features removed in this release.

  - **[Changed]** for changes in existing functionality.
  - **[Security]** to invite users to upgrade in case of vulnerabilities.
  - **[Deprecated]** for once-stable features removed in upcoming releases.

How can I minimize the effort required?

Always have an `Unreleased` section at the top for keeping track of any changes.

This serves two purposes:

- People can see what changes they might expect in upcoming releases.
- Simplification the release process, change `Unreleased` to version and add new `Unreleased`.

Best Naming Practices

- Use short enough and long enough variable names in different scope of code. Generally:
  - `1` char length for `loop` counter name.
  - `1` word for `condition` or `loop` variable name.
  - `1`/`2` words for `function` name.
  - `2`/`3` words for `class`, `struct`, `enum` or `union` name.
  - `3`/`4` words for `global` variable name.
- Do not use too long variable names (e.g. 50 chars).
  - long names will bring ugly and hard-to-read code.
  - may not work on some compilers because of character limit.
- Use specific names for variables.
  - `value`, `equals`, `temp` or `data` are not valid names for any case.
- Decide and use one natural language for naming.
  - e.g. mixed using English and German names will be inconsistent and unreadable.
- Use meaningful names.
  - specify the exact action(what to do) or information, e.g. createPasswordHash.
- Use meaningful names for variables.
  - Variable name must define the exact explanation of its content.
- Do not use the same variable in the different contexts or for different purposes.
  - make it more simplicity for understandability and maintainability.
- Do not use reverse-name-meaning.
  - e.g. `bool is_enable;` is good then `bool is_not_enable;`.
- Do not use non-ASCII chars in variable names.
- Use meaningful names for function parameters.
- BE CONSISTENT.

Best Coding Practices(C/C++)

Usage of Layering

Layering is the primary technique for reducing complexity in a system. A system should be divided into layers. Layers should communicate between adjacent layers using well defined interfaces. When a layer uses a non-adjacent layer then a layering violation has occurred.

A layering violation simply means we have dependency between layers that is not controlled by a well defined interface. When one of the layers changes code could break. We don't want code to break so we want layers to work only with other adjacent layers.

Sometimes we need to jump layers for performance reasons. This is fine, but we should know what we are doing and document it appropriately.

Coding Tips

```
// this is better
if(f())
{
    do_some_thing();
}

// this is not good
if(FAIL != f())
{
    do_other_thing();
    ...
}
```

- Usually Avoid Embedded Assignments

```
d = (a = b + c) + r; // do you realy need to do it like this?
a = b + c;
d = a + r;           // This will be good and clear enough

// there is no other way, and this is good enough
while(EOF != (c = getchar()))
{
    do_some_thing();
}
```

Expressions

- Do not depend on the order of evaluation for side effects
- Use parentheses for precedence of operation
- Do not depend on the order of evaluation of subexpressions
- Do not depend on the order in which side effects take place
- Do not read uninitialized memory
- Do not dereference **null** pointers
- Do not access a variable through a pointer of an incompatible type
- Do not modify constant objects
- Do not compare padding data
- Do not ignore values returned by functions

- Do not assume the size of a structure is the sum of the sizes of its members
- Ensure pointer arithmetic is used correctly
- Use `sizeof` to determine the size of a type or variable
- Do not make assumptions regarding the layout of structures with bit-fields
- Use braces for the body of an `if`, `for`, or `while` statement
- Perform explicit tests to determine **success**, **true** and **false**, and **equality**
- Beware of integer promotion when performing bitwise operations on smaller integer types

Integers

- Ensure that unsigned integer operations do not wrap
- Ensure that integer conversions do not result in lost or misinterpreted data
- Ensure that operations on signed integers do not result in overflow
- Ensure that division and remainder operations do not result in divide-by-zero errors
- Use correct integer precisions
- Use bitwise operators only on unsigned operands
- Avoid performing bitwise and arithmetic operations on the same data
- Evaluate integer expressions in a larger size before comparing or assigning to that size
- Converting a pointer to integer or integer to pointer
- Do not shift an expression by a negative number or a greater number of bits

Floating Point

- Do not use floating-point variables as loop counters
- Ensure that floating-point conversions are within range of the new type

Characters and Strings

- Do not attempt to modify string literals
- Guarantee that storage for strings has sufficient space for character data and the **null** terminator
- Do not pass a non-null-terminated character sequence to a library function that expects a string
- Cast characters to `unsigned char` before converting to larger integer sizes
- Arguments to character-handling functions must be representable as an `unsigned char`
- Do not confuse narrow and wide character strings and functions
- Use pointers to `const` when referring to string literals

Memory Management

- Do not access freed memory
- Free dynamically allocated memory when no longer needed
- Allocate and copy structures containing a flexible array member dynamically
- Only free memory allocated dynamically

- Allocate sufficient memory for an object
- Allocate and free memory in the same module, at the same level of abstraction
- Store a new value(e.g. `NULL`) in pointers immediately after `free()`
- Immediately cast the result of a memory allocation function call
- Beware of zero-length allocations
- Avoid large stack allocations
- Define and use a pointer validation function

Application Programming Interfaces

- Functions should validate their parameters
- Avoid laying out strings in memory directly before sensitive data
- Functions that read or write an array should take an argument to specify target size

Miscellaneous

- No **magic** numbers, use `enum`, `const` or macros.
- Notice the difference of `#if XX`, `#ifdef XX` and `#if defined(XX)`.
- The easiest way commenting out large code blocks is using `#if 0`.
- Do not use floating-point variables where discrete values are needed.
- Using a float for a loop counter is a great way to shoot yourself in the foot.
- Always test floating-point numbers as <= or >=, never use an exact comparison == or `!=`.
- Compile cleanly at high warning levels
- Use comments consistently and in a readable fashion
- Detect and remove dead code
- Detect and remove code that has no effect or is never executed
- Detect and remove unused values
- Do not introduce unnecessary platform dependencies
- Use robust loop termination conditions
- Ensure that control never reaches the end of a non-void function
- Do not use deprecated or obsolescent functions

Emoji and Markdown

Emoji

Make your text funny by emoji!

For quick reference, see: quick-ref-list of emoji in markdown syntax.

- [1] Emoji Cheat Sheet

Markdown

Make your text easy to write and modify later.

For quick reference, see: quick-ref-list of markdown syntax.

- [1] Markdown
- [2] Markdown Style Guide
- [3] Mastering Markdown(GitHub)

Funny Shit

It is time to have something more interesting. :)

- Geek Caricature XKCD