john hopson.com

c/c++ checklist

v 2.2

general

Always strive for **clarity** & **correctness**.

Rules with '**avoid**' or '**favor**' are important, but may be waived if clarity is improved. Rules with '**consider**' are only advice. All other rules are requirements.

Sensibly **document** code as it is originally authored.

Use meaningful naming in every situation, except for local variables in trivially short functions.

Use lower case words, separated with underscores, for everything except macros, which are upper case.

```
sha_encryption.prj
send_email_message()
NUM_DAYS_IN_LEAP_YEAR
```

Spell out potentially misleading abbreviations.

projects

Favor creating a subproject for each major subsection of a large design.

Create **Debug** & **Release** build configurations for each project. Create a **Test** configuration, if applicable.

Create a constant with the build configuration, using the _BUILD suffix.

```
#define  RELEASE_BUILD
```

Put redistributed third-party content under /redist.

```
/redist/linux/drivers/camera
```

Favor this project structure –

_build	build output
data	released data/config files
doc	user help/doc files
redist	third-party content
src	source code
test	test content
tools	build & tool config
makefile	build instructions
version	version information

Avoid build content going anywhere except **_build**.

Place everything but **_build**, under version control.

Place header files alongside source files except for the public API of library projects.

Separate test & production content.

classes/modules

Each source file contains **one class**, & helper classes, or **one interface**, & static helper functions.

Avoid exposing class/module implementation details.

Name files for the interface or class they represent.

Base source files on project templates.

Favor two spaces between #include & file name.

Use brackets for library includes, quotes for project includes.

```
#include  <stdio.h>
#include  "project.h"
```

If include path is necessary, use the **relative** form.

```
#include  "./graphics/gui.h"
```

Limit the **scope** of all elements to the smallest useable level (local *then* module *then* global).

Put only **public** elements in a header file.

In header files, only include other headers required to specify the public interface.

Avoid using 'extern' in a source file.

Favor a **breadth-first** function order (functions of a similar level of abstraction) instead of a depth-first order (functions defined as soon as possible).

If defining a large number of independent utility functions, favor **alphabetical** order.

Place the main() function first, if any.

Define a **virtual** destructor for base classes.

Favor creating a 'to_s()' function for complex structures, and overloading '<<' for each class.

Favor **inline** functions over #define macros.

Use 4 spaces for each indentation level.

Do not use **tabs** or **non-printable** characters.

Avoid lines over 100 columns.

Remove trailing whitespace from every line.

methods/functions

Favor interrogative names for functions that return boolean values

```
are_blast_doors_closed();
is_dad_making_rude_gestures()
```

Use verb-object naming for non-boolean functions

```
create_mmu_relocation_table();
establish_ip_host_connection();
delete_songs_by_air_supply();
```

Use two spaces between the return type & function name

```
bool  fred( uint16 foo,  buffer bar );
```

If too long for one line, column align function prototype on successive lines.

```
uint32  fred(
    word     foo,
    my_type  bar );
```

Use the **const** modifier for any pointer, or referenced data, not modified by the function

```
void  foo( const byte * buffer );
```

Always declare a return type; use 'void' type if none

Favor references over pointers in C++

```
result  send_frame( const buffer&
frame );
```

Declare methods as **const** if they don't alter the state of the instance or class

```
float32  loan::calc_interest()  const
{ ...
```

Functions must produce meaningful results in **all** situations.

Wherever practical, use assertions to document and verify input parameter constraints.

Use vertical & horizontal space to make the code like a readable book, with chapters (functions), paragraphs (related statements) & sentences (individual statements)

```
unrelated_function_call();


/* This 'paragraph' is set off
   from the paragraph above */

for (record = 0;
     record < record_array_length;
     record++)
{
    //  This 'sentence' is
    //  set apart from others
    … code to find record …

    //
}


/*  This 'paragraph' is set off
    from the paragraph above.  */
```

Vertically align short, related function calls

```
if (is_serial_port_ready == true)
{
    foo( console, "Hello"   );
    foo( error,   "Goodbye" );
}
```

Favor straightforward, easy-to-read algorithms. If cleverness is unavoidable, thoroughly document it.

Implement stub functions with '#warning TODO'.

```
int foo(void)
{
    #warning TODO – implement foo()
}
```

operators

Favor a single space before & after each operator

```
three_stooges = larry + curly + moe;
```

Consider **two** spaces separating expression groups.

```
if (foo == true  &&  bar == false)
```

Do not use space around primary operators.

```
serial_port.baud_rate;
serial_port->baud_rate;
```

Do not use space around unary operators.

```
++minutes_since_windows_crash;
```

Only use the '?:' operator in trivial cases.

expressions

Favor no spaces around outer parentheses.

```
if ('0' <= user_input)
```

Favor comparing boolean types to 'true' or 'false'.

Document magic numbers. Initializing with 0, 1 or NULL is an exception.

If an entire conditional does not fit on one line, use multiple vertically-aligned lines.

```
if ('0' <= user_input  &&
    '9' >= user_input)
```

Use the cast operator to make type conversions explicit. Put a space between a cast & its operand.

```
float  size = (uint16) extent;
```

Avoid assignment in conditional expressions.

Do not use parentheses on return expressions.

```
return  foo;
```

statements

Use brackets on single-line statements.

if, **else-if**, **else** statements

```
if (foo >= MAX_FOO)
{
    reduce_foo_count();
}

else if (bar >= MAX_BAR)
{
    reduce_bar_count();
}

else
{
    update_counts();
}
```

for statements

```
for( uint8  irq = 1;  irq < 30;  ++irq
)
{
    init_interrupt( irq );
}
```

or, if too long for one line…

```
for( uint8  num_deleted_records = 0;
            num_deleted_records < 200;
          ++num_deleted_records )
{
    delete_next_record();
}
```

Favor one statement per line.

Only put **loop** conditions in the 'for' statement.

Avoid the comma operator.

Consider **two** spaces after semicolons.

do-while statements

```
do
{
    hit_on_eva_mendes();
}
while( is_eva_running_away  ==  no );
```

switch statements

```
switch( meaning_of_life )
{
  case  self_created:
    mark_user_as_existentialist();
    break;

  case  greatest_happiness:
    mark_user_as_utilitarian();
    break;

  default:
    suggest_pholosophy_text();
    break;
}
```

Consider **two** spaces between 'case' & the constant.

try-catch-finally statements

```
try
{
    do_something();
}

catch( user_exception e )
{
    do_exception_handler();
}

finally
{
    do_cleanup_handler();
}
```

Avoid **goto**.

data

Use names that describe the purpose in the problem domain.

Only use variables for their named purpose.

Favor interrogative statements for boolean names

```
is_file_open, is_database_corrupt,
does_dolly_parton_sleep_on_her_back
```

Use noun phrases for classes & variable names.

```
window_handle, vibration_table
```

Use types that indicate locigal **size** & **signedness**.

8-bit	signed	**int8**
	unsigned	**uint8/byte**
16-bit	signed	**int16**
	unsigned	**uint16/word16**
32-bit	signed	**int32**
	unsigned	**uint32/word32**
64-bit	signed	**int64**
	unsigned	**uint64/word64**
boolean		**bool**
size		**size**

Use two spaces between 'typedef' & the type.

```
typedef  uint23  (* my_type )[];
```

Comment range limits & units of measurement.

```
/* The velocity of an unlaiden
   African or European swallow
   Range 0-200 feet/sec  */
```

Comment array bounds .

Use two spaces between variable type & name.

```
uint16  array_size = 0;
```

Favor **declaring** variables at first use.

Favor **initializing** variables as they are declared.

```
uint16  array_size = 0;
```

Declare each variable on a **separate** line unless the variables have a common purpose.

```
uint32  height, width;
uint32  velocity;
```

Vertically align declarations.

```
uint32    velocity;
Location  current_location;
```

Favor spaces before & after an array index.

```
uint16  foo[ SIZE_OF_FOO ];
```

Initialize pointers with a value or NULL.

```
user_data  * user_data = NULL;
```

Do not use local variable names that override declarations at higher levels.

Limit the scope of all variables to the smallest useable level (local *then* module *then* global).

Do not decorate names with type information.

Avoid global variables.

Hexadecimal numbers use a small 'x' & upper case 'A'-'F'.

```
0x3A2FC3
```

directives

Favor constant variables to #define directives in C++.

```
const int fred = 0;
```

Use uppercase alphanumeric characters in constant names, separating words with an underscore.

```
#define  CONSTANT_DEFINITION  1
```

Use two spaces between '#define' & its value.

Column-align grouped constant values, favoring two spaces after the longest name in the group.

```
#define  CONSTANT         1
#define  LONGER_CONSTANT  0
```

comments

Comments should clearly & succinctly describe how the code is helping achieve the larger goals of the function, module and/or class.

```
/*  Be sure all deleted records are
    marked as stale so garbage coll-
    ector can release that space  */

for( old_record = 0;
     old_record < num_old_records;
     old_record++ )
```

Favor '//' style on single-line comments.

Favor two spaces between comment delimiter ('//' or '/*') & start of comment.

Favor correct English with proper punctuation.

Indent comments at the same indentation level as the code they describe.

```
/*  This comment at the same
    indentation level as the
    code it describes  */

if( mass > CRITICAL_MASS )
{
    // So is this one

    prepare_for_armegeddon();
}
```

Favor adding a blank line after block comments.

Comment anything that might appear confusing to maintainers, such as intricate or obscure algorithms.

Comment the reason for workarounds, & provide URL(s) that describe the need.

Use TODO warnings to mark unfinished sections.

```
#warning TODO - finish db access
```

embedded

Avoid assembly language. Minimize if unavoidable.

Use hand-optimizations only when a performance problem verifiably exists, & then document the optimization (so it can be reassessed later).

Apply 'volatile' to all variables that can be changed outside the compiler's control (e.g. some hardware registers) or outside of the current thread (i.e. shared memory).

Document ISRs with the mask bit, flag, & the means if clearing it, even if it's a side effect.

Use trace(), not printf(), to debug.

Clearly document all hard real time constraints in a **source** file.

Document the startup memory maps in a **source** file at time of creation, even if it may change later.

Trap unused interrupt vectors to a known routine.

Handle all timer timeout scenarios gracefully.

build

Warnings must produce errors in the Release build configuration.

Builds must compile cleanly at the next-to-highest warning level.

Individual warnings may be suppressed if -

The cause is well understood & only that cause can generate the warning
The reason is documented with the suppression code
No reasonable workaround exists

Favor compiling C code with the C++ compiler.

bibliography

1. Standard C: Programmer's Quick Reference, P.J. Plauger & Jim Brodie, Microsoft Press, ©1989 P.J. Plauger & Jim Brodie, ISBN 1-55615-158-6
2. C Style Guide, NASA Software Engineering Lab, Aug 1994, SEL-94-003
3. Code Complete, - A Practical Handbook of Software Construction, Second Edition, Steve McConnell, Microsoft Press, ©2004 Steven C. McConnell, ISBN 978-0735619678
4. MISRA-C:2004, Guidelines for the use of the C language in critical systems, ©2004 The Motor Industry Software Reliability Association

---

hosted at- github.com/johnhopson/checkc