GEM3D Standard Coding Style Guide

Anup Shrestha
anupshrestha@u.boisestate.edu

Dr. Inanc Senocak
senocak@boisestate.edu

July 28, 2016

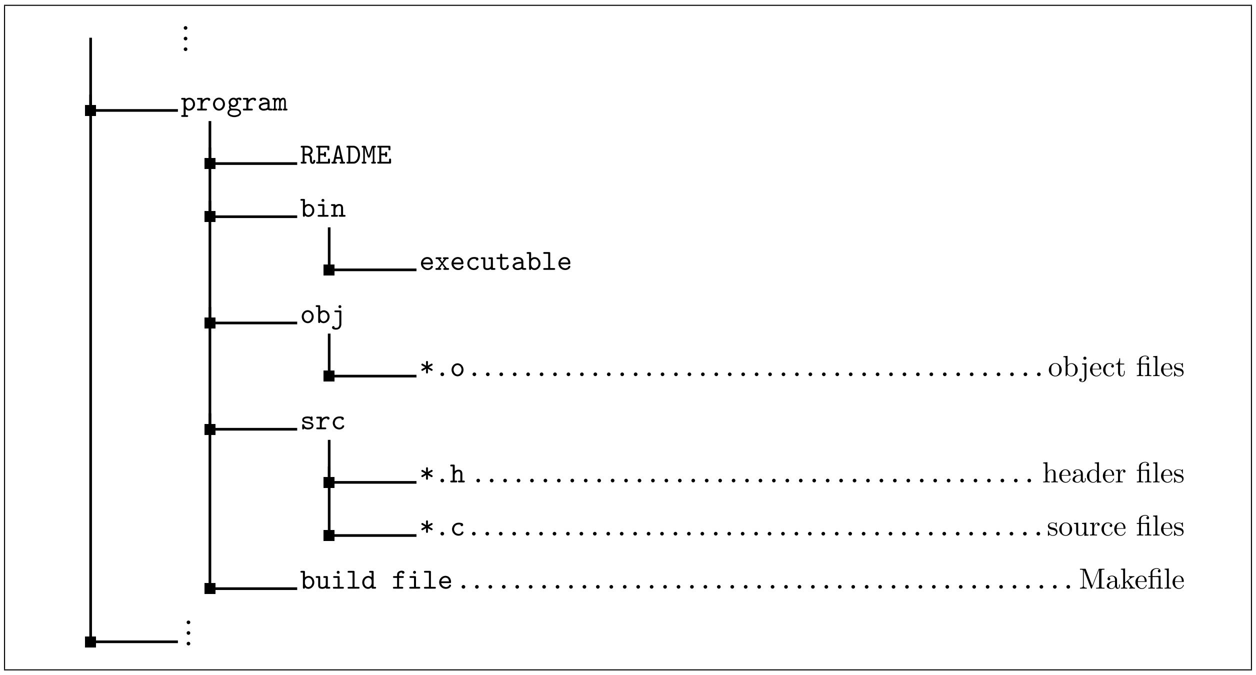
This document discusses recommended practices, coding styles and conventions for writing and organizing a C program. "Good programming style begins with the effective organization of code. By using a clear and consistent organization of the components of your programs, you make them more efficient, readable, and maintainable." - Steve Oualline, C Elements of Style. Coding style and consistency matters for writing efficient code and increasing productivity. A code that is easy to read is also easy to understand, allowing programmers to focus on substance rather than spending time figuring out what the code does. Therefore, a code with "good style" is defined as that which is

- Organized
- Easy to read and understand
- Efficient
- Maintainable and Extensible

The guidelines presented here are based on recommended software engineering techniques, industry standards, recommended practices from the experts in the field and local conventions. Various parts of the guidelines have been extracted from the NASA C Style Guide [1], The C Programming Language [3] and the Linux Kernel Coding Style [2].

1 Program Organization

A C program consists of multiple routines that are grouped together into different files based on their functionality called module or source files. The prototype of the functions of each module file that are shared between different modules are put in a separate file called a header file. Furthermore, usually with large programs there is also a build file (e.g. Makefile) that automates the process of compiling and linking all the files of the program. During compilation, the compiler generates object files for each source file that gets linked together to generate the final executable. Hence, it is easy to see that the number of files in a C program could increase significantly therefore, a proper organization schema is a must to maintain consistency and good style. A recommended schema for organizing a C program is shown in Fig. 1.



program
README
bin
executable
obj
*.o .... object files
src
*.h .... header files
*.c .... source files
build file .... Makefile

Figure 1: Program Organization Schema

The following sections discusses the different files in a C program and recommends a good coding style to follow.

1.1 README File

A README file should be named "README" and succinctly explain what the program does and how the different files are organized. It should contain all the important information about the program as following

- Program name followed by a brief description
- Version, Contact, and License information
- Program Organization
- Requirements and Dependencies
- Instructions for compiling and installing the program
- Usage information for running the program

An example template of a README file is shown in Fig. 2.

1.2 Header Files

A program with good style uses module files to group logically related functions into one file. Header files make it possible to share information between modules that need access to external variables or functions. It also encapsulates the implementation details in the module files. Follow the following guidelines when writing a header file.

- Save the header file with a ".h" extension and use an all lowercase filename that best describes the relation of its content. Avoid names that could conflict with system header files.
- Start header files with an include guard. It prevents the compiler from processing the same header file more than once.
- Use the header filename in all uppercase and append "_H" to define the header.
- Use specialized (e.g., doxygen) comments in order to generate documentation automatically.
- Follow the header file template in Fig. 3 for the prologue.
- Only include other header files if they are needed by the current header file. Do not rely on users including things.
- Group the header file includes logically from local to global and arrange them in an alphabetical order for each group.
- Do not use absolute path in #include directive.
- Use #include <system_file>for system include files.
- Use #include "user_file" for user include files.
- Put data definitions, declarations, typedefs and enums used by more than one program in a header file.
- Only include those function prototypes that need to be visible to the linker.
- End the header file with the following comment on the same line as the end of the include guard.

```
#endif /* END <FILENAME>_H */
```

An example template of a C header file is shown in Fig. 3.

```
Example Program Version #.# MM/DD/YYYY
Copyright (C) <year> <name of author>
Email: <email of author>

Synopsis
----------
A brief introduction and/or overview that explains what the program is.

License
---------
This program is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 2 of the License, or (at your
option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

Program Organization
----------------------
Directory structure similar to Fig. 1.

Requirements/Dependencies
---------------------------
List all requirements including hardware (GPUs, CPU architecture)
and software (operating systems, external modules, libraries).

Installation
--------------
Provide instructions on how to compile and run the code.

Usage
-------
<program_name> <args> [options]

Contributors
--------------
<name of contributor> - <email address>
```

Figure 2: README file template

```
#ifndef FILENAME_H
#define FILENAME_H

/**
 * @file <filename>.h
 * @brief Brief explanation of the header file.
 *
 * Here typically goes a more extensive explanation of what the header
 * defines.
 *
 * @author Last Name, First Name Middle Initial
 * @date DD MON YYYY
 * @see http://website/
 *
 * @note Something to note.
 * @warning Warning.
 * @bug No known bugs.
 *
 */

#include "local1.h"
#include "local2.h"

#include "external.h"

#include <system.h>

/**
 * @brief Brief explanation
 *
 * @param[in]     arg1 explain the argument
 * @param[in,out] arg2 explain the argument
 * @return void
 *
 */
void functionPrototype(int arg1, char *arg2);

#endif /* END FILENAME_H */
```

Figure 3: C header file template

1.3 Source Files

A source file contains the implementation of logically related functions, constants, types, data definitions and declarations [1]. Follow the following guidelines when writing a source file.

- Save the source file with a ".c" extension and use an all lowercase filename that best describes the relation of its content or the same as the header file whose functions it is implementing. If the source file contains the main function then name it "`main.c`".
- Start with the file prologue. Follow the source file template in Fig. 3 for the prologue.
- Then, include header files that are necessary.
- Group the header file includes logically from local to global and arrange them in an alphabetical order for each group.
- Do not use absolute path in #include directive.
- Use #include <system_file>for system include files.
- Use #include "user_file" for user include files.
- Use specialized (e.g., doxygen) comments in order to generate documentation automatically.
- Make the local functions static and follow the guidelines listed in Section 2.6.
- Format the file as described in Section 2.

An example template of a C source file is shown in Fig. 4.

2 Recommended C Coding Style and Conventions

2.1 Naming

Names for files, functions, constants or variables should be descriptive, meaningful and readable. However, over complicating a variable name is also not recommended. For example, a variable that holds a temporary value can always be named 'tmp' instead of 'temporaryCounterVariable' [2]. As a general rule, the scope of a variable is directly proportional to the length of its name. For example, an important variable that is used in most places should have a very descriptive name. Conversely, local variable names should be short, and to the point. A random integer loop counter, should probably be called "i", since this is very common and there is no chance of it being mis-understood [2]. Furthermore, the names of functions, constants and typedefs should be self descriptive, as short as possible and unambiguous. Finally, the names should always be written in a consistent manner as shown in Table 1. Here, for consistency the recommended naming convention to use is *lowerCamelCase* unless specified explicitly.

```
/**
 * @file <filename>.c
 * @brief Brief explanation of the source file.
 *
 * Here typically goes a more extensive explanation of
 * what the source file does.
 *
 * @author Last Name, First Name Middle Initial
 * @date DD MON YYYY
 *
 * @warning Warning.
 * @bug No known bugs.
 *
 */

#include "header.h"

#include <system.h>

/**
 * @brief Brief explanation
 *
 * Add more details here if needed.
 *
 * @param[in]     arg1 explain the argument
 * @param[in,out] arg2 explain the argument
 * @return void
 */
void function(int arg1, char *arg2)
{
    do sth ...
}

/**
 * @brief Brief explanation
 *
 * Add more details here if needed.
 *
 * @param[in] argc command-line arg count
 * @param[in] argv command-line arg values
 * @return int
 */
int main(int argc, char *argv)
{
    return 0;
}
```

Figure 4: C source file template

Table 1: Recommended Naming Convention

Name	Convention
file name	should start with letter and be a noun
function name	should start with letter and be a verb
variable name	should start with letter
constant name	should be in uppercase words separated by underscores
type name	should start with letter and be a noun
enumeration name	should be in uppercase words separated by underscores
global name	prefix using `g_`

2.2 Indentation

Indentation defines an area where a block of control starts and ends. Proper indentation makes the code easier to read. Always use **tabs** to indent code rather than spaces. The rationale being spaces can often be out by one, and lead to misunderstandings about which lines are controlled by if-statements or loops. An incorrect number of tabs at the start of line is easier to spot [3]. Tabs are usually 8 characters however, for scientific code where use of multiple nested for-loops is the norm, having a large indentation is not recommended. Use **4** characters deep indent which will still make the code easy to read and not flushed too far to the right.

2.3 Braces

The placement of the braces is a style choice and different styles have their own benefits. However, for consistency and the benefit of minimizing the number of lines without any loss of readability follow the "K&R" style for brace placement [3]. Their preferred way is to put the opening brace last on the line, and put the closing brace first.

```
if (true) {
        do sth ...
}
```

This applies to all non-function statement blocks (`if, switch, for, while, do`). However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)
{
        ...
}
```

Note that the closing brace is empty on a line of its own, except in the cases where it is followed by a continuation of the statement, i.e., a while in a do-statement or an else in an if-statement, like this:

```
do {                                        if (x == y) {
        ...                                         ...
} while (condition);                        } else if (x > y) {
                                                    ...
                                            } else {
                                                    ...
                                            }
```

Do not unnecessarily use braces where a single statement will do.

```
   if (condition)                              if (condition)
           action();                                   doThis();
                                               else
                                                       doThat();
```

This does not apply if only one branch of a conditional statement is a single statement; in the latter case use braces in both branches:

```
if (condition) {
        doThis();
        doThat();
} else {
        otherwise();
}
```

2.4 Spaces

Spaces should be used to separate components of expressions, to reveal structure and make clear intention. Good spacing reveals and documents programmer's intentions whereas poor spacing can lead to confusion. [3]

- Use a space after keywords like:
  `if, switch, case, for, do, while`.
- Use one space on each side of binary and ternary operators like:
  `= + - < > * / | & ^ <= >= == != ?  :  %`
- Use a space (or newline) after commas `,` and semicolons `;`.
- Do not use space after unary operators like:
  `& * + - ~ !  sizeof typeof alignof __attribute__ defined`

- Do not use space around the primary operators like:
  `->, ., and [].`
- Do not use space before the prefix and after the postfix increment and decrement unary operators like:
  `++ --`
- Do not leave trailing whitespace at the ends of lines.

2.5 Commenting

Comments when used properly can provide important information that cannot be discerned by just reading the code. However, if used improperly it can make code difficult to understand. When writing comments do not try to explain how the code works, let the code speak for itself. In general comments should explain what the code does and possibly why it does it. The preferred style for multi-line comments is specialized, doxygen style comment block, so that documentation can be generated automatically.

```
/**
 * \brief Brief description.
 *        Brief description continued.
 *
 * This is the preferred style for multi-line
 * comments using the doxygen style comment block.
 * Two column of asterisks on the left side of the
 * first line. Then, a single column of asterisks
 * on the left side, ending with almost-blank line.
 *
 * \author Name
 * @see functions()
 */
```

Also, add comments to data throughout the file, where they are being declared or defined. Use one data declaration per line (i.e., no commas for multiple data declarations), leaving space for small comment on each item, explaining its purpose. If more than one short comment appears in a block of code or data definition, start all of them at the same tab position and end all at the same position.

```
void someFunction()
{
        doWork();     /* Does something */
        doMoreWork(); /* Does something else} */
}
```

2.6 Functions

Design functions to be short that does just one thing and does that well. Each function should be preceded by a function prologue that gives a short description of what the function does and how to use it. Avoid duplicating information clear form the code. An example function prologue is shown in Fig 5. The recommended conventions for declaring a function

- The function's return type, name and arguments should be on the same line, unless the number of arguments do not fit on a single line. In that case move the rest of the arguments to the next line aligned with the arguments on the line above.
- The name of the function should be descriptive and follow the naming conventions defined in Table 1.
- Do not default to int; if the function does not return a value then it should be given return type void.
- The opening brace of the function body should be alone on a line beginning in column 1.
- Declare each parameter, do not default to int.
- Comments for parameters and local variables should be tabbed so that they line up underneath each other.
- In function prototypes, include parameter names with their data types.

2.7 Structs, Macros and Enums

- Structs

  Structures are very useful feature in C, that enhances the logical organization of the code and offers consistent addressing [1]. The variables declared in a struct should be ordered by type to minimize any memory wastage because of compiler alignment issues, then by size and then by alphabetical order.

  ```
  struct point3D {
          int x;
          int y;
  ```

```
/**
 * @brief Brief explanation
 *
 * Add more details here if needed.
 *
 * @param[in]     arg1 explain the argument
 * @param[in,out] arg2 explain the argument
 * @return void
 */
void function(int arg1, char *arg2)
{
    do sth ...
}
```

Figure 5: C function template

```
        int z;
}
```

- Macros

  Using the #define preprocessor command to define constants is a convenient technique. It not only improves readability, but also provides a mechanism to avoid hard-coding numbers. Use uppercase letters to name constants and align the various components as shown in the example below.

  ```
  #define NULL  0
  #define FALSE 0
  #define TRUE  1
  ```

  On the other hand avoid using macro functions. They are generally a bad idea with potential side-effects without any advantages.

- Enums

  Enumeration types are used to create an association between constant names and their values. Use uppercase letters to name the enum type and the constants. Place one variable identifier per line and use aligned braces and indentation to improved readability.

```
enum POSITION {
        LOW,
        MIDDLE,
        HIGH
};
```

```
enum POSITION {
        LOW    = -1,
        MIDDLE = 0,
        HIGH   = 1
};
```

3 Automatic Code Formatting

It can be really time consuming and difficult to follow all the standard coding style while writing code. Therefore, it is a good idea to use tools and IDEs that can automate the process of formatting the code based on specific rules. One such tool is called the clang-format. Clang-format has various different options specifying the type of formatting style to be applied to specific parts of the code. More information about the clang-format and the style options can be found at http://llvm.org/releases/3.8.0/tools/clang/docs/ClangFormat.html.

References

[1] Doland, Jerry and Valett, Jon, *C Style Guide.* Software Engineering Laboratory Series, 1994. `http://homepages.inf.ed.ac.uk/dts/pm/Papers/nasa-c-style.pdf`

[2] Torvalds, Linus, *Linux kernel coding style.* Linux Kernel Archives. `https://www.kernel.org/doc/Documentation/CodingStyle`

[3] Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language.* Prentice Hall Inc, 1988. `https://www.kernel.org/doc/Documentation/CodingStyle`