C Style Guide for KF5006

Entity Naming (variables and functions)

1. **Names shall begin with a lower case letter, shall be all lower case and words shall be separated by an underscore (_).**

2. **Use sensible, descriptive names.**

Do not use short cryptic names or names based on internal jokes. It should be easy to type a name without looking up how it is spelt.

Exception: Loop variables and variables with a small scope may have short names to save space if the purpose of that variable is obvious.

3. **Only use English names.**

It is confusing when mixing languages for names. English is the preferred language because of its spread in the software market and because most libraries that have been developed already use English.

4. **Variables with a large scope can have long names, variables with a small scope can have short names.**

Variables that are used for temporary storage or array indices are best kept short. A programmer reading such variables shall be able to assume that its value is not used outside of a few lines of code. Common variable names for short lived, integer variables are `i`, `j`, `k`, `m`, `n` *and for short lived character names are* `c` *and* `d`*. if that variable is to be returned by a function a longer more descriptive name should be used.*

Indentation and Spacing

5. **Braces shall follow "K&R Bracing Style".**

The K&R Bracing Style was first introduced in very popular book “The C Programming Language by Brian Kernighan and Dennis Ritchie”.

For anything other than function definitions the opening brace is placed at the end of the statement and preceded by a space. The closing brace is on a new line and lined up with the related statement e.g.

```
if (string_length > max_length) {
  printf(“The string is too long.\n”);
} else {
  Printf(“The string length is in range.\n”);
}
```

Braces, not used for functions shall be indented 2 columns to the right of the starting position of the enclosing statement or declaration.Statements and declarations between the braces are indented relative to the enclosing statement. The opening brace of a function body is placed on a line of its own and lined up with the function declaration.

```
int heat_regulator(float temperature, float set_point)
{
  const int CORRECTION = 2;
  int adjustment = 0;
  if (temperature > set_point) {
    adjustment = -CORRECTION;
  } else {
    adjustment = CORRECTION;
  }
  return adjustment;
}
```

6. Loop and conditional statements shall always have brace enclosed sub-statements.
The code looks more consistent if all conditional and loop statements have braces.

Even if there is only a single statement after the condition or loop statement today, there might be a need for more code in the future.

7. Braces without any contents may be placed on the same line.
The only time when two braces can occur on the same line is when they do not contain any code.

```
for (int i = 0; i < 1000; ++i) {}
```

8. Each statement shall be placed on a line of its own.
There is no need to make code compact. Putting several statements on the same line only makes the code cryptic to read.

9. Declare each variable in a separate declaration.
This makes it easier to see all variables. It also avoids the problem of knowing which variables are pointers.

```
int* p, i;
```

It is easy to forget that the asterisk belongs to the declared name (p in the above case) and not the type (int), in this case the type is int and `p` *is a pointer to int but* `i` *is declared as an int. The following makes it more clear:*

```
int *p, i;
```

But the following is much better and should be used to adhere to this style guide:

```
int *p;
int i;
```

Note that in C the asterisk can be next to the data type or the variable name, both are equivalent in functionality. This style guide recommends placing the asterisk next to the variable name to make it clear that the variable is a pointer.

10. All binary arithmetic, bitwise and assignment operators and the ternary conditional operator (?:) shall be surrounded by spaces; the comma operator shall be followed by a space but not preceded.

Note: the ternary operator is a shortcut version of an if else that uses ? and returns a value:

```
If (x > y) {
  result = x;
} else {
  result = y;
}
```

Is the same as ->

```
result = (x > y) ? x : y;
```

11. Lines shall not exceed 78 characters.

Even if your editor handles long lines, other people may have set up their editors differently. Long lines in the code may also cause problems for other programs and printers.

12. Do not use tabs.

Tabs make the source code difficult to read where different editing programs treat the tabs differently. The same code can look very differently in different editors so to to avoid this problem use spaces instead.

Comments

13. Comments shall be written in English

It is confusing when mixing language. English is the preferred language because of its spread in the software market and because most libraries that have been developed already use English.

14. Use the following style comments.

The comment styles `///` and `/** ... */` are used by Doxygen and some other code documenting tools. Please use these to adhere to this style guide. The use of these comments will be discussed further when we talk about designing functions.

15. All comments shall be placed above the line the comment describes, and indented identically.

Being consistent on placement of comments removes any question on what the comment refers to.

16. Use `#if 0, #endif` instead of `/* ... */` to comment out blocks of code.

The code that is commented out may already contain comments which then terminate the comment block and can cause lots of compile errors, or other harder to find errors.

```
int heat_regulator(float temperature, float set_point)
{
    const int CORRECTION = 2;
    int adjustment = 0;
    #if 0
    /// this block of code is disabled for debugging
    if (temperature > set_point) {
        adjustment = -CORRECTION;
    } else {
        adjustment = CORRECTION;
    }
    /// this is the end of the disabled block
    #endif
    return adjustment;
}
```

17. Every function shall have a comment that describes its purpose.

```
/** This function compares the current temperature with
 *  a set point temperature and returns an adjustment
 *  value.
 */
int heat_regulator(float temperature, float set_point)
{
    const int CORRECTION = 2;
    int adjustment = 0;
    if (temperature > set_point) {
        adjustment = -CORRECTION;
    } else {
        adjustment = CORRECTION;
    }
    return adjustment;
}
```

Files

18. File names should be treated as case sensitive and should be self-documenting.
Self-documenting means that you should try and use file names that describe what the file contains e.g.:

string.h

Just from the file name and extension we have a good idea that the file will contain function prototypes that are designed to work with strings.

19. C source files shall have extension "`.c`".

20. C header files shall have extension "`.h`".

21. Binary executable files shall be named after the source file that includes the `main()` function, but without the `".c"`.
This just makes it easier for us to keep track of which binary executable was produced from which source file. For example the source file get_links.c should produce the binary executable get_links using:

gcc –Wall get_links.c –o get_links

22. Header files must have include guards (header guards).
The include guard protects against the header file being included multiple times e.g.

```
#ifndef FILE_H
#define FILE_H
/// header file contents go here
#endif
```

The pre-processor directive:

#ifndef

Checks to see if the name that follows has been defined, if it hasn't the contents of the header file are processed and the name FILE_H is defined with the pre-processor directive:

#define

A subsequent inclusion of the same header file will conclude that the name FILE_H has been defined already and so the contents of the header file will be ignored (as they have already been included). This stops header files being included twice. Alternatively the pragma once directive may be used if the compiler supports it:

#pragma once

23. The name of the macro e.g. FILE_H used in the include guard shall have the same name as the file (excluding the extension) followed by the suffix "_H".

The pre-processor uses the term macro to describe something which has been given a name (defined).

24. System header files shall be included with <> and your headers with " ".

25. Put `#include` directives at the top of files.

Having all `#include` directives in one place makes it easy to find them.

26. Do not use absolute directory names in `#include` directives.

The directory structure may be different on other systems.

Declarations

27. Provide names of arguments/parameters in function prototypes (declarations).

Argument/Parameter names are useful to document what the function argument is used for. The name should be the same in the prototype declarations and the definition of the function.

28. Do not use global variables.

Global variables are variables that are declared outside of any function and so are accessible globally. They are initialised when the program starts whether used or not. If global variables are using other global variables for their initialisation there may be a problem those that are used not initialised yet. The initialisation order of global variables in different executable files is not defined.

Statements

29. Never use gotos.

`goto` statements break the structured of your code and the over use of these make code very difficult to follow through and hence debug.

30. Do not use `break` in loops.

A `break` statement is a `goto` statement in disguise and makes code less readable.
A break statement is only acceptable in switch statements where it is part of the statement structure.

31. Do not use `continue` in loops.

A `continue` statement is a `goto` statement in disguise and makes code less readable.

32. Try to only have one return in a function

It can be confusing when more than one return statement is used in a function. Having only one exit point from a function makes it easy to have a single place to check the logic of the functions code. For example when debugging It is useful to have a single exit point from a function where you can put a single breakpoint or trace output(more on this when we start to use the gdb debugger). It is sometimes necessary to introduce a result variable to carry the function return value to the end of the function. This is an acceptable compromise for structured code.

33. All switch statements shall have a default label.

Even if there is no action for the default label, it shall be included to show that the programmer has considered values not covered by case labels.

Other Issues

34. Try not to use literal values other than 0, 1 and '\0'.

Use constants instead of literal values to make the code consistent and easy to maintain. The name of the constant is also used to document the purpose of the variable:

```
const int NUMBER_OF_LEGS = 2;
const float PI = 3.142;
const char START = "<a";
```

Note: it is customary to name constant that are declared at the beginning of a function using capital letters.

35. Do not rely on boolean values in conditions.

Sometimes it is convenient, when you know that a function or a variable will produce a Boolean value, to forego the usual comparison that uses a Boolean operator e.g.

```
/// this conditiontests the FILE pointer
/// for a valid FILE handle
if (encrypted_file) {
    …
}
```

For readability It is better to be explicit in the comparison:

```
/// this is better, more explicit in what is being tested
if (encrypted_file != NULL) {
    …
}
```

Note: Most organisations will have many more rules and guidelines than this, but we only have two semesters to cover a lot of material so we have kept the guidelines to a minimum. We have included the most common styles, used across style guides, which we have come across.