UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CSE013E: INTRODUCTION TO PROGRAMMING IN C

Style Guidelines

Lines & Spacing

1. Lines should be limited to 100 characters (tabs count as 4 characters) and this includes comments. Example code adhering to this style guideline is shown in examples: [8, 9] at the end of this document.
2. Lines that are too long should be split at sensible spots.
    - After a comma in a function argument list
    - At a reasonable point in a string (not in the middle of a word). [8]
    - After operators in an expression. [9]
3. Split lines should be indented at least one level further in, ideally being right-aligned with the line above. [8, 9]
    - Indentation pass the tab-point of the parent line should only use spaces so the code remains readable even with different tab sizes.
4. All code blocks should increase their indentation by 1 tab character as they're nested. [4]
5. Code within `#ifdef`'s and the like should be indented to match where they would be inserted into the code if possible to increase readability. [16]
6. Expressions within for-loops should follow standard spacing guidelines but with the addition that there is no space before the semicolons, but 1 space afterwards. [10, 15]
7. Macros should be split into multiple lines if they include complex logic. [3]
8. All non-unary operators should have spaces separating them from their operands. [5, 6, 7]
9. Commas should always be followed by a space. [15]
10. Array initialization with curly-braces should not include spaces between the braces and the array element values. Parentheses should also not include spaces inside the parentheses. [4]
11. Switch statements should include their case conditions at the same indentation as the switch itself. Therefore the actual code should sit one indentation in from the switch statement. [14]
12. Struct member operators ('`.`', '`->`') should not be surrounded by spaces. [4]
13. There should be no whitespace at the ends of lines.
14. Hexadecimal numbers should be written in all upper-case. [5]
15. The asterisk used when referring to pointers ('*') should be directly next to the variable name with no spacing. [4, 7, 8]
16. Extraneous spacing on a line should be kept to a minimum, so non-indentation whitespace should be limited to a single space.
17. Lines should only contain a single statement, so generally one semi-colon per line. [4, 5, 7]
    - Special case is for-loop declarations. [3, 10, 15]

Headers

1. Header files should include header guards. [12]

- Header guards protect code from being included in files multiple times, which can cause errors.
- Header files therefore follow the template:

  ```
  #ifndef X
  #define X
  HEADER_FILE_CODE
  #endif
  ```

  , where X is an underscored and uppercased version of the header's filename, ex. CircularBuffer.h should have header guard constants named _CIRCULAR_BUFFER_H_

2. Headers should only `#include` the necessary files for the header to compile properly. All headers that are only necessary for the corresponding C-file should be included directly in that file.

Braces

1. Curly-braces ({}) should surround every non-empty control statement that can have them (if, else, while, etc) even if the body has only a single line of code. [13]
2. Curly-braces should always exist on the same line as the statement their modifying. [4, 10, 11, 13, 18]
3. Opening curly-braces for functions should exist on the line below the function header. [17, 18]

Comments

1. Comments should be written as inline comments "`//`" unless declaring large sections of code or functions. [4, 11]
2. Comments can exist on the same line as code, but this needs to respect the 100 character limit for a single line. This is also reserved for initial variable declarations and useful for documenting struct members inline. [4]
3. Block comments ("`/* */`") should be used to comment general areas of code, such as the location of header files, macros, and function prototypes. The left hand side of each line within the block comment should start with a space and an asterisk ('*') and the ending line should be a space, asterisk, forward slash ("` */`"). [15]
4. Function and macro-function block comments should follow the same rules as regular block comments but include an extra asterisk on the first line so it should look like ("`/**`"). [1, 12]
5. Function and macro-function block comments should include the valid input for the function, expected output, error handling, and anything else relevant such as whether the return variable has been `malloc()`ed and will need to be manually `free()`d. [18]
6. Comments should not describe what the code is doing unless it is at a very high level. Just describing the code is redundant as the code should already be readable. Comments should focus on why this code is this way or noting any other peculiarities of it. [13, 14]

Naming

1. Functions should be named in a NounVerb capacity such as `MatrixPrint()` or `MatrixScalarAdd()`. [18]
2. Functions that are contained within the same module should be prefixed with the module name or something else that makes sense. [12]
   - For example the MatrixMath.h library uses 'Matrix' as the prefix for all of its functions.
3. Functions, structs, and enums should follow an upper-camelcase convention where all words and

acronyms are squished together without underscores and the start of each one is capitalized.

- Acronyms should count as a single word and as such should only have their first letter capitalized.
- Good examples are `MatrixPrint()`, `CanMessage` (CAN is an acronym), and `ListInsertAfter()`.

4. Variable names should follow a lower-camelcase convention which only differs from the above convention in that the first word is NOT capitalized.
5. Variable names should avoid long and overly similar names where possible (like the pair `arrayIndexUpperBound` and `arrayIndexLowerBound`) as they are hard to distinguish.
6. Variable names should be named to be indicative of their use. They should only be used for that purpose unless memory use is a factor.
7. Macro names should be all uppercase and use underscores for word separation. [1, 2, 3]
8. If temporary variables are required, name them something that involves tmp, temp, or temporary, e.g. `tmp` or `tempCalculation`. Ideally place this code in its own code block so that the variable is out of scope later and so doesn't need to be unique. In this way you avoid conflicts and having to use numbering with the variables.
9. Single-letter variables should be lower-case. These are only valid for counter variables for loops. For integers use the standard convention of `i`, `j`, `k`, `l`, `m`, or `n`.

Scope & Organization

1. All structures in C should exist in the smallest scope necessary to provide the desired API
    - Don't expose structures, variables, functions that other code shouldn't use
2. Functions should be limited to the scope they're necessary in. Functions internal to a module should NOT be declared in the header and exposed to other programs.
3. Global variables that are only necessary in a single C-file should be declared `static` and thus limited to module scope.
4. Code should be logically grouped into small segments by using empty lines as a separator. [4]
5. C-files for libraries should be organized as follows:
    1. File comments
        - name, software license, broad description, etc.
    2. Included files
        - Including the corresponding header file for this file.
        - Grouped into system files (those included with <>) and user files (those included with “”)
    3. Data type definitions
    4. Internal macro declarations
        - These macros would be only for use within this C-file. Macros that are part of the library should be placed in the header file.
    5. Global and external variable declarations
    6. Functions definitions
6. C-files for the file with `main()` should be organized as follows:
    1. File comments
        - name, software license, broad description, etc.
    2. Included files
        - Grouped into system files (those included with <>) and user files (those included with “”)

3. Data type definitions
4. Internal macro declarations
    - These macros would be only for use within this C-file. Macros that are part of the library should be placed in the header file.
5. Global and external variable declarations
6. Internal function prototypes
    - These prototypes would be only for functions defined and used only within this C-file.
7. `main()`
8. Internal function definitions
    - These functions are only for use within `main()` and other functions within this file. Functions for external use should be moved into an additional module.

Macros

1. Macros should use parentheses around any input arguments. [1, 3]
2. Any magic numbers, numerical constants with special meanings, used within the code should be instead referred to via a macro. [2, 14]
    - This will generally exclude loop termination conditions.
    - Usually this refers to any constants used in the code.
3. Macros should never affect control flow! This negatively affects both readability and debugging.
    - Avoid `break`, `continue`, and `return` within a macro.
4. Parentheses should also surround the entire macro if they're not just constants. [1]

Code examples

1. Example 1

```
/**
 * Calculates the length of the third side of a triangle given the other two sides based on
 * the Pythagorean theorem. Note that x and y will be evaluated twice in the macro so be careful.
 */
#define PY(x, y) (sqrt((x) * (x) + (y) * (y))
```

2. Example 2

```
// The constant pi that relates to circles and trigonometry.
#define PI 3.14159
```

3. Example 3

```
#define LIST_LOOP(cons, listp)                                       \
       for ((cons) = (listp); !NILP (cons); (cons) = XCDR (cons)) {  \
              if (!CONSP (cons)) {                                    \
                     signal_error ("Invalid list format", (listp));  \
              }                                                       \
       }                                                             \
```

4. Example 4

```c
// A point specified in circular coordinates.
struct Point {
       double Angle;  // Units in radians
       double Radius; // Units in meters
};

struct Point origin = {0.0, 0.0};

if (zoomValue > 8.0) {
       origin.Angle = M_PI;
       origin.Radius = 1;
}

struct Point *originReference = &origin;
originReference->Radius = 0.0;
originReference->Angle = 0.0;
```

5. Example 5

```c
int debugSpeed = (++i << 5) ^ 0x7FFE;
```

6. Example 6

```c
unsigned int bitToSet = 1 << (3 | ((message.buffer & 1) << 3));
```

7. Example 7

```c
uint32_t output[3];

output[1] = ((uint32_t)msg.payload[3]) << 24;
output[1] |= ((uint32_t)msg.payload[2]) << 16;
output[1] |= ((uint32_t)msg.payload[1]) << 8;
output[1] |= (uint32_t)msg.payload[0];

output[2] = ((uint32_t)msg.payload[7]) << 24 |
            ((uint32_t)msg.payload[6]) << 16 |
            ((uint32_t)msg.payload[5]) << 8;
             (uint32_t)msg.payload[4];

output[3] = (((uint32_t)msg.validBytes) << 16);
```

8. Example 8

```c
char *HelloString = “Are you still there? I see you. Hello? Critical error. Sorry. I don't hate you”
                                                                                  “. Why? No hard feelings.”;
```

9. Example 9

```
unsigned long BitToSet = (1 << (3 | ((message.buffer & 1) << 3))) & (++i << 5) ^ 0x7FFE) |
                                                    ((unsigned long)msg.payload[3]) << 24;
```

10. Example 10

```
int i;
for (i = 0; (i < lim - 1) && ((c = getchar()) != EOF) && (c != '\n'); ++i) {
       s[i] = c;
}
```

11. Example 11

```
// Transmit the 6-byte message via UART1.
int bytesToTransmit = 6;
int i = 0;
while (i < bytesToTransmit) {
       PutsUART1((unsigned int)txMsg[i++]);
}
```

12. Example 12

EcanFunctions.h:

```
#ifndef __ECAN_FUNCTIONS_H__
#define __ECAN_FUNCTIONS_H__

/**
 * Pops the top message from the ECAN1 reception buffer. It returns a tCanMessage struct with the
 * older message data.
 */
struct CanMessage Ecan1Receive();

#endif // __ECAN_FUNCTIONS_H__
```

EcanFunctions.c:

```
#include “EcanFunctions.h”
#include “Buffer.h”

// Keep a circular buffer for the received CAN messages.
static struct CircBuffer ecan1RxBuffer;

struct CanMessage Ecan1Receive()
{
      return BufferReadFront(&ecan1RxBuffer);
}
```

13. Example 13

```
// Set transmission errors in first array element.
if (C1INTFbits.TXBO) {
      errors[0] = 3;
} else if (C1INTFbits.TXBP) {
      errors[0] = 2;
} else if (C1INTFbits.TXWAR) {
      errors[0] = 1;
}

// Set reception errors in second array element.
if (C1INTFbits.RXBP) {
      errors[1] = 2;
} else if (C1INTFbits.RXWAR) {
      errors[1] = 1;
}
```

14. Example 14

```
// A little Boolean emulation for C
#define TRUE  1
#define FALSE 0

// Control the mux for the command inputs between autonomous or manual mode.
switch (runningMode) {
default:
case AutonomousMode:
       driveSwitch = FALSE;
       rudderSwitch = FALSE;
       break;
case ManualMode:
       driveSwitch = TRUE;
       rudderSwitch = TRUE;
       break;
}
```

15. Example 15

```
/*
 * Include standard libraries.
 */
#include  <stdio.h>
#include <math.h>

int i, j;
for (i = 0, j = 0; i < 10; ++i, j = pow(i, 2)) {
       printf(“%d^2 = %d\n”, i, j);
}
```

16. Example 16

```
#ifdef __PIC24F__
#include "p24fxxxx.h"
```

```
#else
#error This code was designed for the PIC24F on the Explorer16 development board.
#endif
```

17. Example 17

```
int main()
{
	printf("Hello World!\n");
	return 0;
}
```

18. Example 18

```
/**
 * Integer division function. Valid input ranges for 'a' and 'b' are any valid integers. If 'b' is 0
 * then the return value will be 0.0 as this is an invalid computation.
 */
double IntegerDivide(int a, int b)
{
	if (b == 0.0) {
		return 0.0;
	} else {
		return (a / b);
	}
}
```