laSalle ENG

Universitat Ramon Llull

Style Guide

Authors:
Ferran Obiols Jordan
Xavier Canaleta LLampallas
Arturo Roversi Cahiz

Reviewers:
Aaron Jeriel Pica Pardos
David Vernet Bellet
Francesc Teixido Navarro

31st October 2024
v1.0

Contents

1 Introduction

When programming, it is very important to make the code readable and understandable, not only for others but also for oneself.

Following structuring rules will make our code more reusable for future modifications or expansions, achieving a time saving in its development.

For these reasons, this style guide has been created, which through simple rules will allow both students and instructors to quickly follow the code that is being developed/evaluated.

It should be noted that this guide is not a C language standard set by the IEEE; it is just a set of rules selected from various sources in the business world and that are quite common in the software industry. Knowing this guide does not guarantee knowing how to program in C since there are many and very diverse guides like this one. But it does provide a good introduction to the world of real programming and facilitates the interpretation of other style guides.

2 Files

In any application, it is mandatory to correctly modularize the code. Distributing functions and grouping them by functionality into different files with self-explanatory names.

2.1 Naming Convention

For naming files, we will follow the patterns in the following table:

Type	Extension	Example
C source code	*.c	main.c
Header files	*.h	globals.h
Make files	Makefile	makefile
Shared library	*.so	lib.so

2.2 Headers

All files must include a header that indicates the following fields:

1. File name.
2. Purpose or use of the module (including arguments if the main function is in this file).
3. Author's name.
4. Date of last modification.

```
1  /*************************************************
2  *
3  * @File:
4  * @Purpose:
5  * @Author:
6  * @Date:
7  *
8  *************************************************/
9
10     ... ...
```

2.3 Source Code Files

Source code files should contain the following sections:

1. Header
2. Inclusions
3. Functions

Inclusions

In the inclusion section, we will put all the includes of the necessary libraries for that document. In the case of using your own header files, the system library includes should go only in the header file, and in the source code file only the inclusions of your own header files.

Example:

```
... header ...

// System
#include <stdio.h>
#include <stdlib.h>
    .
    .
    .
// Own
#include "main.h"
    .
    .
    .

... other sections ...
```

Functions

Functions are the last section of a source code file. They don't have to follow any specific order, but it is recommended to group them by similarity or purpose to reduce the subsequent search time.

This section does not cover the declaration of functions. For that, consult section 5.

Example:

```
int main(void){

  sayHello();

  return 0;
}

void sayHello(){

  printf("Hello");

}
```

Example Source File

```
/*********************************************
*
* @File:
```

```
4  * @Purpose:
5  * @Author:
6  * @Date:
7  *
8  ***********************************************/
9
10 #include "interface.h"
11
12 int main(void){
13
14   sayHello();
15
16   return 0;
17 }
18
19 void sayHello(){
20
21   printf("Hello");
22
23 }
```

2.4 Header Files

All header files must have a mechanism to prevent them from being included more than once. This is achieved by using the following structure, modifying __FILE_H__ to the name of our file, e.g., __MAIN_H__, __HEADERS_H__, etc.

```
1 #ifndef __FILE_H__
2 #define __FILE_H__
3
4     ... CONTENT ...
5
6 #endif
```

Content

Header files should use the following order in their content:

1. File header
2. Inclusions
   - First, system includes (if any are needed)
   - Then, project-specific includes (if any are needed)
3. Constants, defines, and macros
4. Global definitions
5. Prototypes

Example Header File

```c
#ifndef __MAIN_H__
#define __MAIN_H__

/*********************************************
*
* @File:
* @Purpose:
* @Author:
* @Date:
*
*********************************************/

// System Includes
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>

// Project Includes
#include "interface.h"

// Constant Declarations
#define TPORT 8000
#define THERE "so.housing.salle.url.edu"
#define LPORT 8275
#define LIBER "vela.salle.url.edu"

// Custom Type Definitions
typedef struct{
  int nCount;
  char aacUsers[256][8];
}list;

typedef struct{
  int nWin;
  char acUmsg[1000];
}message;

// Global Variables
int nNum;
int nTotal;

// Prototypes
void exitProgram();
void salarm();
```

```
46 void childCode();
47 void parentCode(char *argv);
48 int main(int argc, char *argv[]);
49
50 #endif
```

3 Defines, Macros, and Constants

The names of defines, macros, constants, etc., must be self-explanatory. They should inherently contain an explanation of their use or purpose.

Constants should be defined in uppercase (both those defined by '#define' and those defined using 'const').

If they are composed of multiple words, they should be separated by the character "_".

It is advisable to use defines to replace the use of numerical values in the code.

Example:

Replace:

```
if (fNumberPi == 3.14){
    ...
}
```

With:

```
#define PI                  3.14
#define DAYS_OF_WEEK        7

if (fNumberPi == PI){
    ...
}
```

4 Variables

4.1 Variable Naming Conventions

The names used for variables must be self-explanatory, so that the name itself indicates the use or purpose of the variable.

A very common practice, and the one that should be used to name variables in the course, is to use a series of prefixes to identify the type of variable in a simple and quick way. This type of nomenclature is called Hungarian notation (devised by Charles Simonyi, chief architect at Microsoft). These prefixes are added to the variable name without separating them in any way.

Naming table:

Type	Prefix	Example
void	v	void vEmptyVariable;
char	c	char cKeyPressed;
int	n	int nNumberOfCells;
long	l	long lTotalUnits;
short	sh	short shSmallVariable;
float	f	float fUnitPrice;
double	d	double dMinimumAngle;
byte or unsigned char	by	unsigned char byMask;
word or unsigned int	w	unsigned int wFlags;
* (pointer)	p	int *pnValueArray;
& (reference)	r	float &rfMaximumAngle;
(array)	a	double adMaximumRanges[3];
enum (enumeration)	e	EBool eResult;
typedef	t	typedef int tStudentGrade;
struct	st	struct stAirport ... ;

For character strings, one might think that the correct way to define them is, for example:

```
1 // Incorrect definition of a character string
2 char acFileName[256];
```

But to be able to identify them, since they are widely used, we will use the prefix `s`. For example:

```
1 // Correct definition of a character string
2 char sFileName[256];
```

On the other hand, we will differentiate the declaration of a character string using ***[]***, from the definition of a character pointer ***char ** ** that would contain the memory address where this character string begins. For this particular case, we will use the following nomenclature:

```
1 // Declaration of a character pointer
2 char *psPointerToString;
```

Using this nomenclature somewhat contradicts the rules established above, but it can help to differentiate variables of character pointers (individual characters) from variables of pointers to character strings (Strings).

Another case not shown in the previous table is the use of the `unsigned` prefix to declare unsigned variables, which—except for the types `char` and `int` that, when adding the `unsigned` prefix, are

declared as `byte` and `word` respectively—in other cases, we will simply add the prefix `u` to the already assigned prefix of the variable. Example:

```
unsigned long ulUnsignedLongVariable;
unsigned int wUnsignedIntVariable;
```

Exceptions

Exceptionally, for the implementation of counters used in loops, it will be allowed to use numerical variables called `i`, `j`, etc., as long as it does not harm the understanding of the code (and these variables are used only within the `for` loop).

```
int i;
for (i = 0; i < 100; i++) { ... }
```

Use of Variables

All variables must be declared in the most restricted scope possible. This leads to more efficient use of the stack and a reduction of possible errors due to "collateral" effects.

It is advisable that each variable is declared separately and that all variables are initialized in their own declaration.

```
int nCounter         = 0;
int nNumberOfRows    = 0;
int nNumberOfColumns = 0;
int nTotalElements   = 0;
```

Local Variables

Local variables will be placed at the beginning of the function declaration following the guidelines described above, and with an indentation of one tab.

```
void testFunction() {

    int nNumberOfRows    = 0;
    int nNumberOfColumns = 0;
    char cKeyboardChar   = ' ';

    ... function code ...

}
```

Global Variables

In general, ***GLOBAL VARIABLES SHOULD NOT BE USED*** unless completely unavoidable.

The existence of global variables goes against code comprehension and encapsulation, and can cause unexpected "collateral" effects (if a function changes the value of the global variable in an uncontrolled way) that lead to very difficult to identify errors.

In the unavoidable case of their use, a `g` must be added before the previously specified nomenclature.

Example:

```
1 int gnCounter          = 0;
2 int gnNumberOfRows     = 0;
3 int gnNumberOfColumns = 0;
4 int gnTotalElements    = 0;
```

And these variables, as already specified above, must be declared in header files.

5 Functions

5.1 Function Headers

All function definitions must include a descriptive header that indicates the following fields.

1. Function name.
2. Purpose or use of the function.
3. Function arguments, indicating whether they are input or output, and the purpose or use of each.
4. Function return value, indicating the possible return values and their meanings.

```
/*****************************************************************
*
*   @Name: requestMemory
*   @Def: Given the size of the letters ...
*   @Arg:  In: nLetters = number of letters that will form the string
*          Out: psString = character string with ...
*   @Ret: Returns 1 if operation successful, 0 otherwise.
*
*****************************************************************/

int requestMemory(int nLetters, char *psString) {

    ... function code ...

}
```

5.2 Function Declaration

- Function names must be self-explanatory. They should give a clear idea of their purpose.
- Function names will follow the lowerCamelCase nomenclature, where the first letter of each word is capitalized except for the first letter, and all others are in lowercase. Example: exampleOfLowerCamelCase.
- Functions should not be longer than 45 lines.
- Once the function is declared, the braces will open at the same level as the function declaration and will close below the last line of code at the same indentation level as the function declaration. In the case of the opening brace, there must be a mandatory space between the closing parenthesis and the opening brace.

Example:

```
1 void freeMemory (char *psWord) {
2
3     ... function code ...
4 }
5
6 char* requestMemory (int nLetters) {
7
8     ... function code ...
9 }
```

5.3 Whitespace

In the function declaration, the arguments received by the function will follow exactly the following pattern: function(arg1, arg2, arg3, ...). Example:

```
1 void someFunction (int nLetters, char cACharacter, int nWordCount) {
2
3     ... function code ...
4 }
```

6 Statements

All statements will be indented one level relative to the function/statement that contains them, which is equivalent to one tabulation or four spaces. Similarly, all content within a statement will be indented one tabulation or four spaces relative to the statement.

Regarding the placement of braces in statements, they will follow the same pattern as in function declarations. The opening brace will be placed at the same level as the statement declaration, and the closing brace will be placed below the last line of code, at the same indentation level as the statement declaration. In the case of the opening brace, there must be a mandatory space between the closing parenthesis and the opening brace.

6.1 Comparisons

When using comparisons between a variable and a constant, the constant should be indicated as the first member of the comparison. This is done to prevent possible errors (very difficult to identify) by indicating an assignment instead of a comparison.

Thus, the following statement would be less correct:

```
1 // Less correct way
2 if (nData == 10) {
3     ...
4 }
```

Instead, the correct way would be:

```
1 // Correct way
2 if (10 == nData) {
3     ...
4 }
```

6.2 Assignments

Multiple assignments can lead to erroneous actions; it can (erroneously) be understood that certain variables are equivalent (in which case they would be unnecessary) and do not contribute to code readability. Therefore, they should not be used.

Incorrect:

```
1 // Incorrect multiple assignment
2 nXUpper = nYUpper = (nScreenWidth - nWindowWidth) / 2;
```

Correct:

```
1 // Correct assignment
2 nXUpper = (nScreenWidth - nWindowWidth) / 2;
3 nYUpper = nXUpper;
```

6.3 Ternary Operator

It is advisable to use `if` statements instead of the ternary operator `?`. This ternary operator, typical of C, does not offer advantages over the `if` statement, which is more widely known, extensively used, and leads to more readable code.

```
// Use of the ternary operator (not recommended)
nReturnValue = (bStatusCorrect ? doSomething() : doSomethingElse());

// Use of if... else... statement, more correct
if (bStatusCorrect) {
    nReturnValue = doSomething();
} else {
    nReturnValue = doSomethingElse();
}
```

6.4 `for` Statement

The `for` statement will be used to define a loop in which a variable is incremented constantly in each iteration, and the loop's termination is determined by a constant expression.

```
for (i = 0; i < nMaxTurns; i++) {
    ...
}
```

6.5 `while` and `do...while` Statements

The `while` statement will be used to define a loop in which the termination condition is evaluated at the beginning of the loop.

The `do...while` statement will be used to define a loop in which the termination condition is evaluated at the end of the loop.

At the beginning of a `while` or `do...while` loop, the control expression must have a clearly defined value to prevent possible determinations or operational errors.

```
int nVariableCount;
...
// Code that does not modify nVariableCount
...

// Initialize the loop counter
nVariableCount = 0;

while (MAX_COUNT < nVariableCount) {
    ...
}
```

6.6 `switch` Statement

In `switch` statements, you must always include the `default` option and the `break` in all branches.

A skeleton of a `switch` statement:

```
// Example of using switch
switch (nCondition) {
    case 1:
        ...
```

```
5           break;
6
7       case 2:
8           ...
9           break;
10
11      default:
12          ...
13  }
```

6.7 `goto` Statement

In general, ***DO NOT USE `goto` STATEMENTS***.

The `goto` statements and other similar jump statements that break the code structure cause a nonlinear alteration in the program's flow. They seriously compromise the integrity of the code. They may seem to offer a quick solution to specific problems, but they entail many possible collateral problems, unpredictable and difficult to determine.

7 Comments

Throughout the document, we have already explained how to make some of the comments; in this section, we will revisit some of the previously covered cases and introduce some new ones.

7.1 Files

As we have seen, at the beginning of each file, you should include the following code fragment that describes various information about the file.

```
/*****************************************************************
*
* @File:
* @Purpose:
* @Author:
* @Date:
*
*****************************************************************/

    ... File Content ...
```

7.2 Functions

Similar to the previous section, before each function, we must justify the definition of that function by stating the function's name, the definition of its functionality, the arguments it receives (both input and output), and the value it returns.

```
/*****************************************************************
*
*   @Name: requestMemory
*   @Def: Given the size of the letters ...
*   @Arg:  In: nLetters = number of letters that will form the string
*          Out: psString = character string with ...
*   @Ret: Returns 1 if the operation is successful, 0 otherwise.
*
*****************************************************************/

int requestMemory(int nLetters, char *psString){

    ... function code ...

}
```

7.3 Lines

Line comments serve to clarify the function that a code fragment performs. But for a comment to be correct does not mean that it has to describe the obvious; rather, it should explain the reason or meaning of the line. For example:

Example of a useless comment:

```
1 // We set variable i to 0
2 i = 0;
```

Example of a useful comment:

```
1 // We initialize the variable to reset the loop and avoid possible
    errors in the iterations.
2 i = 0;
```

Format

Comments should always be placed before the relevant code fragment, never beside it. A comment can be used to comment on more than one line; there is no need to write a comment for each line of code. You can provide a small explanation that summarizes what a series of lines do.

Single line:

```
1 i = 0;  // incorrect
2
3 // correct
4 i = 0;
```

Multiple lines:

```
1 /*
2     This is a comment
3     spanning multiple lines
4 */
5 i = 0;
```

8 Indentation

8.1 Functions

In the case of functions, whether it is the `main` function or any other, they will not be indented relative to the margin of the page. However, their content is always indented by one tabulation relative to the level where the function is declared.

```

int main(){

// This comment is at an incorrect level

  // This comment is at the correct level (1 tabulation or 4 spaces)

    // This comment is also at an incorrect level

}
```

8.2 Statements

As already stated in the section on statements, all statements will be indented one tabulation relative to the function/statement that contains them, or the equivalent of one tabulation, which will be four blank spaces. Similarly, all content within a statement will be indented by one tab or four spaces relative to the statement.

Example of catastrophic indentation:

```

int main(){

// Incorrect declaration
int i;

// Incorrect statement
for(i=0; i<MAX_COUNT; i++) {
printf("%d \\n", i);
}

}
```

Example of correct indentation:

```

int main(){

  // Correct declaration
  int i;
  int j;

  // Correct statement
```

```
9     for(i=0; i<MAX_COUNT; i++) {
10      printf("%d\\n", i);
11
12
13      for(j=0; j<MAX_COUNT; j++) {
14        printf("%d\\n", j);
15
16      }
17
18    }
19
20  }
```

9 Whitespace

9.1 Operators

As can be seen in all the examples in this document, there will always be a blank space between an operator (=, <, >, |, &, %) and a variable or constant to facilitate reading. Examples:

```

i = 10;

if (10 == nValue) {
    ...
}

fResult = 122 % 2;
```

9.2 Statements

9.2.1 for

In the case of the ***for*** statement, apart from the spaces corresponding to the operators, there will also be a space after each ";" that separates the different arguments that this statement receives.

```

for (i = 0; i < MAX_VAL; i++) {
    ...
}
```

9.3 Functions

In the declaration of functions, the arguments that the function receives will follow exactly the following pattern: function(arg1, arg2, arg3, ...). Example:

```

void someFunction(int nLetters, int nWordCount) {

    ... function code ...
}
```

10 Bibliography

[1] Hungarian Notation, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvsgen/html/hunganotat.asp, Charles Simonyi, Microsoft Corporation.

[2] C++ Programming Style, Tom Cargill, ed. Addison Wesley Professional.

[3] C++ Programming Style, http://www.spelman.edu/~anderson/teaching/resources/style/, Scoot D.Anderson.

[4] Manual De Estilo C/C++, http://www.geocities.com/webmeyer/prog/estilocpp/, Oscar Valtueña García, V1.1.0, 2005

[5] The Definitive Guide to GCC Second Edition,William von Hagen, ed. Apress.