laSalle ENG

Universitat Ramon Llull

Guía de Estilo

Autores:
Ferran Obiols Jordan
Xavier Canaleta LLampallas
Arturo Roversi Cahiz

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

14 de setembre de 2012
v1.0

Índex

1 Introducción **3**

2 Archivos **4**
2.1 Convención de nombres 4
2.2 Cabeceras 4
2.3 Archivos de Código fuente 5
2.4 Archivos de cabecera 7

3 Defines, macros y constantes **10**

4 Variables **11**
4.1 Nomenclatura de las variables 11

5 Funciones **14**
5.1 Cabeceras de las funciones 14
5.2 Declaración de Funciones 14
5.3 Espacios en blanco 15

6 Sentencias **16**
6.1 Comparaciones 16
6.2 Asignaciones 16
6.3 Operador ternario 17
6.4 Sentencia for 17
6.5 Sentencias While y do while 17
6.6 Sentencia Switch 18
6.7 Sentencia goto 18

7 Comentarios **19**
7.1 Archivos 19
7.2 Funciones 19
7.3 Lineas 20

8 Indentación **21**
8.1 Funciones 21
8.2 Sentencias 21

9 Espacios en Blanco **23**
9.1 Operadores 23
9.2 Sentencias 23
9.2.1 for 23
9.3 Funciones 23

10 Bibliografia **24**

Referències **24**

1 Introducción

A la hora de programar es muy importante hacer que el código sea legible y comprensible, no solamente para otras personas, sino para uno mismo.

Seguir unas normas de estructuración hará que nuestro código sea más reusable para posteriores modificaciones o ampliaciones, consiguiendo un ahorro de tiempo en el desarrollo del mismo.

Por estos motivos se ha creado esta guía de estilo, que mediante sencillas reglas permitirá tanto a alumnos como instructores hacer un seguimiento rápido del código que se está desarrollando/evaluando.

Se debe tener en cuenta que esta guía no es un estándar del lenguaje C marcado por la IEEE, es solo un conjunto de reglas seleccionadas de diversas fuentes del mundo empresarial y que son bastante comunes en el mundo del software. Conocer esta guía no garantiza saber como programar en C ya que de guías como esta hay muchas y muy diversas. Pero sí permite tener una buena introducción en mundo de la programación real y facilita la interpretación de otras guías de estilo.

2 Archivos

En cualquier aplicación es obligatorio modular correctamente el código. Repartiendo las funciones y agrupándolas por funcionalidad en archivos diferentes con un nombre auto explicativo.

2.1 Convención de nombres

Para el nombramiento de los archivos seguiremos los patrones de la siguiente tabla:

Tipo	Extensión	Ejemplo
Código fuente en C	*.c	main.c
Archivos de encabezados	*.h	globals.h
Archivos de "make"	Makefile	makefile
Biblioteca compartida	*.so	lib.so

2.2 Cabeceras

Todos los archivos deben incluir una cabecera que indique los siguientes campos:

1. Nombre del archivo.
2. Finalidad o uso del módulo (incluyendo los argumentos si en este archivo se encuentra la función main).
3. Nombre del autor.
4. fecha de última modificación.

```
1
2  /*********************************************
3  *
4  * @Archivo:
5  * @Finalidad:
6  * @Autor:
7  * @Fecha:
8  *
9  *********************************************/
10
11
12    ... ...
```

2.3 Archivos de Código fuente

Los archivos de código fuente deben contener los siguientes apartados:

1. Cabecera
2. Inclusiones
3. Funciones

Inclusiones

En la sección de inclusiones pondremos todos los includes de las librerías necesarias para ese documento. En el caso de estar utilizando archivos de cabecera propios, los includes de librerías del sistema deberán ir únicamente en el archivo de cabecera y en el archivo de código fuente sólo las inclusiones de los archivos de cabecera propios.

Ejemplo

```
1
2 ... cabecera ...
3
4 //Sistema
5 #include <stdio.h>
6 #include <stdlib.h>
7   .
8   .
9   .
10 //Propias
11 #include "main.h"
12   .
13   .
14   .
15
16
17 ... otras secciones ...
```

Funciones

Las funciones son el último apartado de un archivo de código fuente, no tienen que seguir ningún orden en concreto pero es recomendable agruparlas por similitud o finalidad para reducir el posterior tiempo de búsqueda.

En este apartado no se cubre la declaración de la funciones. Para ello consultar el apartado 5.

Ejemplo:

```
1
2 int main (void){
3
```

```
4   diHola();
5
6   return 0
7 }
8
9 void diHola(){
10
11   printf("Hola");
12
13 }
```

Ejemplo archivo fuente

```
1
2 /*********************************************
3 *
4 * @Archivo:
5 * @Finalidad:
6 * @Autor:
7 * @Fecha:
8 *
9 *********************************************/
10
11 #include "interface.h"
12
13
14 int main (void){
15
16   diHola();
17
18   return 0
19 }
20
21 void diHola(){
22
23   printf("Hola");
24
25 }
```

2.4 Archivos de cabecera

Todos los archivos de cabecera tienen que tener un mecanismo para impedir que sean incluidos más de una vez. Esto se consigue utilizando la siguiente estructura, modificando __ARCHIVO_H__ por el del nombre de nuestro archivo __MAIN_H__ , __CABECERAS_H__ , etc.

```
1
2 #ifndef __ARCHIVO_H__
3 #define __ARCHIVO_H__
4
5    ... CONTENIDO ...
6
7 #endif
```

Contenido

Los documentos de encabezados deben usar el siguiente orden en su contenido:

1. Cabecera del archivo
2. Inclusiones
    - Primero los includes del sistema (si hace falta alguno)
    - Después los includes propios del proyecto (si hace falta alguno)
3. Constantes, defines y macros
4. Definiciones globales
5. Prototipos

Ejemplo archivo de cabeceras

```
1
2
3 #ifndef __MAIN_H_
4 #define __MAIN_H_
5
6
7 /*********************************************
8 *
9 * @Archivo:
10 * @Finalidad:
11 * @Autor:
12 * @Fecha:
13 *
14 *********************************************/
15
16 //Includes del Sistema
17 #include <stdio.h>
18 #include <fcntl.h>
19 #include <string.h>
20 #include <stdlib.h>
21
22 //Includes propios
23 #include "interface.h"
24
25 //Declaración de Constantes
26 #define TPORT 8000
27 #define THERE "so.housing.salle.url.edu"
28 #define LPORT 8275
29 #define LIBER "vela.salle.url.edu"
30
31 //Definición de tipos propios
32 typedef struct{
33   int nCount;
34   char aacUsuaris[256][8];
35 }llista;
36
37 typedef struct{
38   int nWin;
39   char acUmis[1000];
40 }missatge;
41
42
43 //Variables globales
44 int nNum;
45 int nTotal;
```

```
46
47 //Prototipos
48 void surt();
49 void salarm ();
50 void codiFill();
51 void codiPare(char *argv);
52 int main(int argc, char *argv[]);
53
54 #endif
```

3 Defines, macros y constantes

Los nombres de defines, macros, constantes, etc... tienen que ser auto explicativos. En sí mismos tienen que contener una explicación de su uso o finalidad.

Las constantes se tienen que definir en mayúsculas (tanto las definidas por #define como las definidas utilizando const).

Si se componen de varias palabras se deberían separan por el carácter "_".

Se aconseja el uso de defines para sustituir el uso de valores numéricos en el código.

Ejemplo:

Substituir:

```
1
2 if (fNumeroPi == 3.14){
3    ...
4 }
```

Por:

```
1
2 #define PI                    3.14
3 #define DIAS_DE_LA_SEMANA     7
4
5 if (fNumeroPi == PI){
6    ...
7 }
```

4 Variables

4.1 Nomenclatura de las variables

Los nombres utilizados para las variables tienen que ser auto explicativos. De manera que en el propio nombre esté la indicación del uso o finalidad de las variables.

Una práctica muy habitual y la que se deberá utilizar para nombrar las variables en la asignatura, es utilizar una serie de prefijos para poder identificar el tipo de variable de una forma sencilla y rápida. Este tipo de nomenclaturas se denomina notación húngara (ideada por Charles Simonyi, arquitecto jefe de Microsoft). Estos prefijos se añaden al nombre de la variable, sin separarlos de ninguna forma.

Tabla de nomenclatura:

Tipo	Prefijo	Ejemplo
void	v	void vVariableVacía;
char	c	char cTeclaPulsada;
int	n	int nNumeroCeldas;
long	l	long lTotalUnidades;
short	sh	short shVariablePequeña
float	f	float fPrecioUnitario;
double	d	double dAnguloMinimo;
byte o unsigned char	by	unsigned char byMascara
word o unsigned int	w	unsigned int wFlags
* (puntero)	p	int *pnMatrizValores;
& (referencia)	r	float &rfMaximoAngulo;
(array)	a	double adRangosMaximos[3];
enum (enumeracion)	e	EBoole eResultado;
typdef	t	typedef int tNotaAlumno;
struct	st	struct stAeropuerto ... ;

Para el caso de cadenas de caracteres, podríamos pensar que la forma correcta de definirlas es, por ejemplo:

```
1  //Definición incorrecta de una cadena de caracteres
2  char acNombreArchivo[256];
```

Pero para poder identificarlas, ya que se usan de manera extendida, utilizaremos el prefijo s. Por ejemplo:

```
1  //Definición correcta de una cadena de caracteres
2  char sNombreArchivo[256];
```

Por otro lado, diferenciaremos la declaración de una cadena de caracteres mediante el uso de *[]*, de la definición de un puntero a carácter ***char ** ** que contendría la dirección de memoria donde empieza esta cadena de caracteres. Para este caso en particular se utilizara la siguiente nomenclatura:

```
1  //Declaración de un puntero a carácteres
2  char *psPunteroAUnString;
```

Utilizando esta nomenclatura contradecimos un poco la normas establecidas anteriormente, pero puede ayudar a diferenciar las variables de punteros a caracteres (caracteres individuales) a las variables de punteros a cadenas de caracteres (Strings).

Otro caso que no se muestra en la anterior tabla, es el caso de prefijo unsigned para declarar variables sin signo, que a excepción de los tipos char y int que al añadir el prefijo unsigned se declaran como byte y word respectivamente. En los otros casos simplemente se añadirá el prefijo u al ya asignado prefijo propio de la variable. Ejemplo:

```
1  unsigned long ulVariableDeTipoUnsignedLong;
2  unsigned int wVariableDeTipoUnsignedInt;
```

Excepciones

Excepcionalmente, para la implementación de contadores utilizados en bucles, estará permitido la utilización de variables numéricas llamadas i, j, ... siempre que no perjudique la comprensión del código (y estas variables se utilicen solamente dentro del bucle del for).

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

Utilización de variables

Todas las variables hay que declararlas en el ámbito más restringido posible. Esto lleva a una utilización más eficiente de la pila y a una reducción de los posibles errores por efectos "colaterales".

Es aconsejable que cada variable se declare de forma separada y que todas las variables sean inicializadas en la propia declaración de las mismas.

```
1 int nContador       = 0;
2 int nNumeroFilas    = 0;
3 int nNumeroColumnas = 0;
4 int nTotalElementos = 0;
```

Variables locales

Las variables locales se situaran al principio de la declaración de la función siguiente las guías anteriormente descritas. Y con una indentación de un tabulado.

```
1
2 void funcioDeTest(){
3
4   int nNumeroFilas    = 0;
5   int nNumeroColumnas = 0;
6   char cLletraTeclat = ' ';
7
8   ... código de la función ...
9
10 }
```

Variables globales

En general, ***NO SE DEBEN UTILIZAR VARIABLES GLOBALES*** salvo en caso totalmente inevitable. La existencia de variables globales atenta contra la comprensión del código y su encapsulacion, además de que puede provocar efectos "colaterales" inesperados (si una función varía el valor de la variable global de forma no controlada) que desembocan en errores muy difíciles de identificar.

En el caso inevitable de su utilización, se deberá añadir una g delante de la anterior nomenclatura especificada.

Ejemplo:

```
1
2 int gnContador        = 0;
3 int gnNumeroFilas     = 0;
4 int gnNumeroColumnas = 0;
5 int gnTotalElementos = 0;
```

Y estas variables como ya se ha especificado anteriormente, deben de estar declaradas en los archivos de cabeceras.

5 Funciones

5.1 Cabeceras de las funciones

Las definiciones de todas las funciones tienen que incluir una cabecera descriptiva que indique los siguientes campos.

1. Nombre de la función.
2. Finalidad o uso de la función.
3. Argumentos de la función, indicando si son de entrada o salida, y la finalidad o uso de cada uno de ellos.
4. Retorno de la función. Indicando los posibles valores de retorno y el significado de los mismos.

```
1
2 /*****************************************************************
3 *
4 * @Nombre: pedirMemoria
5 * @Def: Dado el tamaño de las letra ...
6 * @Arg: In: nLetras = numero de letras que formaran el string
7 *       Out: psString = cadena de caracteres con ...
8 * @Ret: devuelve 1 si operación satisfactoria, 0 en caso contrario.
9 *
10 *****************************************************************/
11
12 int pedirMemoria(int nLetras, char *psString) {
13
14   ... código de la función ...
15
16 }
```

5.2 Declaración de Funciones

- Los nombres de las funciones tienen que ser auto explicativos. Tienen que dar una idea clara de cual es su finalidad.
- Los nombres de las funciones seguirán la nomenclatura lowerCamelCase, donde la primera letra de cada una de la palabras ira en mayúscula a excepción de la primera letra, y todas la demás irán en minúscula. Ejemplo: ejemploDeLowerCamelCase.
- La funciones no deberían medirá más de 45 lineas.
- Una vez declarada la función, las llaves se abrirán a la misma altura que la declaración de la función y se cerraran debajo de la ultima línea de código al mismo nivel de indentación que la declaración de la función. En el caso de la abertura, obligatoriamente habrá un espacio en blanco entre el paréntesis de cierre y la apertura de la llave.

Ejemplo:

```
1
2 void liberarMemoria (char *psPalabra) {
3
4   ... código de la función ...
5 }
6
7 char* pedirMemoria (int nLetras) {
8
9   ... código de la función ...
10 }
```

5.3 Espacios en blanco

En la declaración de las funciones, los argumentos que recibe la función, seguirán exactamente el siguiente patrón: función (arg1, arg2, arg3, ...). Ejemplo:

```
1
2 void algunaFuncion (int nLetras, char cUnCaracter, int
    nCountDePalabras) {
3
4   ... código de la función ...
5 }
```

6 Sentencias

Todas la sentencias, estarán tabuladas con una tabulación respecto a la función/sentencia que las contenga o el equivalente a una tabulación que serán cuatro espacio en blanco. Del mismo modo todo el contenido de una sentencia estará indentado con un tabulador o cuatro espacios respecto la sentencia.

En cuanto a la apertura de las llaves en las sentencias, seguirán el mismo patrón que en la declaración de las funciones. Las llaves se abrirán a la misma altura que la declaración de la sentencia y se cerraran debajo de la ultima línea de código, al mismo nivel de indentación que la declaración de la sentencia. En el caso de la abertura, obligatoriamente habrá un espacio en blanco entre el paréntesis de cierre y la apertura de la llave.

6.1 Comparaciones

Cuando se utilicen comparaciones entre una variable y una constante, hay que indicar como primer miembro de la comparación la constante. Esto se hace así para prevenir posibles errores (muy difíciles de identificar) por indicar una asignación en vez de una comparación.

Así la siguiente sentencia sería poco correcta:

```

if (nDato == 10) {
   ...
}
```

En cambio lo correcto sería realizarlo así:

```

if (10 == nDato) {
   ...
}
```

6.2 Asignaciones

Las asignaciones múltiples pueden dar lugar a actuaciones erróneas, se puede entender (de manera errónea) que ciertas variables son equivalentes (en cuyo caso sobrarían) y no da legibilidad al código. Por lo tanto no se deben utilizar.

Incorrecto

```

nXSuperior = nYSuperior = (nAnchoPantalla - nAnchoVentana)/2;
```

Correcto:

```


nXSuperior = (nAnchoPantalla - nAnchoVentana)/2;
nYSuperior = nXSuperior;
```

6.3 Operador ternario

Es aconsejable utilizar sentencias if en vez del operador ternario ?. Este operador ternario ?, propio de C, no presenta ventajas frente a la sentencia if, que es más conocida, de uso extendido y da lugar a un código más legible.

```
1
2 // Uso del operador ternario (desaconsejado)
3 nValorRetorno = (bStatusCorrecto ? hacerUnaCosa() : hacerOtra());
4
5 //Uso de sentencia if... else..., mas correcta
6 if (bStatusCorrecto) {
7   nValorRetorno = hacerUnaCosa();
8 }else{
9   nValorRetorno = hacerOtra();
10 }
```

6.4 Sentencia for

La sentencia for se usará para definir un bucle en el que una variable se incrementa de manera constante en cada iteración y la finalización del bucle se determina mediante una expresión constante.

```
1
2 for (i=0; i<nMaximoVueltas; i++){
3   ...
4 }
```

6.5 Sentencias While y do while

La sentencia while se usará para definir un bucle en el que la condición de terminación se evalúa al principio del bucle.

La sentencia do...while se usará para definir un bucle en el que la condición de terminación se evaluará al final del bucle.

Al comenzar un bucle while o do...while la expresión de control debe tener un valor claramente definido, para impedir posibles determinaciones o errores de funcionamiento

```
1
2 int nVariableCount;
3 ...
4 //Codigo que no modifica nVariableCount
5 ...
6
7 // Inicializamos el contador del bucle
8 nVariableCount = 0;
9
10 while (MAXIMO_CONTADOR < nVariableCount) {
11   ...
12 }
```

6.6 Sentencia Switch

En las sentencias switch hay que incluir siempre la opción default y el break en todas las ramas.

Un esqueleto de una sentencia switch:

```
1
2 // Ejemplo de uso del switch
3 switch (nCondicion){
4   case 1:
5   ...
6   break;
7
8   case 2:
9   ...
10  break;
11
12  default:
13  ...
14 }
```

6.7 Sentencia goto

En general ***NO USAR SENTENCIAS GOTO***.
Las sentencias goto y otras sentencias de salto similares que rompen la estructura del código provocan una alteración no lineal en el flujo del programa. Atentan seriamente contra la integridad del código. Aparentemente pueden suponer una solución rápida a problemas puntuales, pero conllevan muchos posibles problemas colaterales, imprevisibles y de difícil determinación.

7 Comentarios

A lo largo del documento ya se ha relatado como realizar algunos de los comentarios, en este apartado se volverán exponer algunos de los casos ya expuesto y alguno de nuevo.

7.1 Archivos

Como ya hemos visto al principio de cada archivo se tendrá que poner el siguiente fragmento de código que describirá información diversa sobre el archivo.

```
1
2  /*****************************************************************
3  *
4  * @Archivo:
5  * @Finalidad:
6  * @Autor:
7  * @Fecha:
8  *
9  *****************************************************************/
10
11
12   ... Contenido del archivo ...
```

7.2 Funciones

Parecido al apartado anterior, antes de cada función tendremos que justificar la definición de dicha función exponiendo, el nombre de la función, la definición de su funcionalidad, los argumento que recibe tanto de entrada como de salida, y el valor que retorna.

```
1
2  /*****************************************************************
3  *
4  * @Nombre: pedirMemoria
5  * @Def: Dado el tamaño de las letra ...
6  * @Arg: In: nLetras = numero de letras que formaran el string
7  *       Out: psString = cadena de caracteres con ...
8  * @Ret: devuelve 1 si operación satisfactoria, 0 en caso contrario.
9  *
10 *****************************************************************/
11
12 int pedirMemoria(int nLetras, char *psString){
13
14   ... código de la función ...
15
16 }
```

7.3 Lineas

Los comentarios de línea sirven para esclarecer la función que desarrolla un fragmento de código. Pero para que se un comentario correcto no significa que se tenga que describir lo obvio sino más bien el motivo o significado de la línea, por ejemplo:

Ejemplo de comentario inútil:

```
1
2 //Ponemos la variable i a 0
3 i = 0;
```

Ejemplo de comentario útil:

```
1
2 //Inicializamos la variable par hacer un reset del bucle y evitar
     posibles errores en la iteraciones.
3 i = 0;
```

Formato

Los comentarios se colocaran siempre antes del fragmento en cuestión, nunca al lado. Un comentario puede servir para comentar más de una linea no hace falta poner un comentario para cada linea de código, se puede exponer un pequeña explicación que resuma lo que hacen un seguido de lineas.

Una linea:

```
1
2   i = 0;  //incorrecto
3
4   //correcto
5   i = 0;
```

Múltiples lineas:

```
1
2   /*
3     Esto es un comentario
4     de múltiples lineas
5   */
6   i = 0;
```

8 Indentación

8.1 Funciones

En el caso de las funciones, ya sea la función main o cualquier otra, no se tabularan respecto el margen de la pantalla. Pero su contenido siempre esta a una tabulación respecto el nivel en que se ha declarado la función.

```
1
2 int main(){
3
4 //Este comentario esta a un nivel incorrecto
5
6   //Este comentario esta al nivel correcto (1 tabulación o 4 espacios)
7
8     //Este comentario también esta a un nivel incorrecto
9
10 }
```

8.2 Sentencias

como ya se ha expuesto en la sección de sentencias, todas la sentencias estarán tabuladas con una tabulación respecto a la función/sentencia que las contenga o el equivalente a una tabulación, que serán cuatro espacio en blanco. Del mismo modo todo el contenido de una sentencia estará indentado con un tabulador o cuatro espacios respecto la sentencia.

Ejemplo de indentación catastrófica:

```
1
2 int main(){
3
4 //Declaración incorrecta
5 int i;
6
7 //sentencia incorrecta
8 for(i=0; i<MAX_COUNT ;i++) {
9 printf("%d \\n", i);
10 }
11
12 }
```

Ejemplo de indentación correcta:

```
1
2 int main(){
3
4   //Declaración correcta
5   int i;
6   int j;
7
8   //sentencia correcta
```

```
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 Espacios en Blanco

9.1 Operadores

Como se puede apreciar en todos los ejemplos de este documento, siempre entre un operador = < > | & % y una variable o constante siempre habrá un espacio en blanco para facilitar la lectura. Ejemplos

```

i = 10;

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

fResultat = 122 % 2;
```

9.2 Sentencias

9.2.1 for

En el caso de la sentencia ***for***, a parte de los espacios correspondientes a los operadores, también habrá un espació detrás de cada ";"que separa los diferentes argumentos que recibe esta sentencia.

```

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

9.3 Funciones

En la declaración de las funciones, los argumentos que recibe la función, seguirán exactamente el siguiente patrón: función (arg1, arg2, arg3, ...). Ejemplo:

```

void algunaFuncion (int nLetras, int nCountDePalabras) {

   ... código de la función ...
}
```

10 Bibliografia

[1] Hungarian Notation, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvsgen/h
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.