Guia d’estil de de programació en C

La Salle – Universitat Ramon Llull

(última actualització: octubre 2019)

Índex

Introducció

Actualment, en el món de la programació, és molt important escriure codi que sigui llegible y comprensible. Tant per un mateix, com per la resta de persones que vulguin consultar el nostre codi.

Un codi ben estructurat, seguint una sèrie de normes acordades, sempre facilitarà la comprensió dels diferents elements a l'autor i als potencials lectors. Així, es facilitarà també la reutilització i el manteniment del codi ja escrit.

L'objectiu d'aquest document és marcar una normativa que faciliti la comprensió del codi que escriguin tant alumnes com professors.

Cal dir que aquesta guia d'estil no és un estàndard del llenguatge C, sinó que es tracta d'un conjunt de normes seleccionades a partir de les recomanacions fetes per diverses fonts.

El fet d'utilitzar aquesta guia d'estil facilitarà a l'usuari introduir-se en el món de la programació real. Si es segueixen aquestes recomanacions, l'usuari adoptarà unes normes comunes d'escriptura de codi, però no el dotarà de coneixement exhaustiu del llenguatge C.

1. Variables

1.1. Nomenclatura de les variables

El nom que li donem a una variable és molt important. Facilitarà la lectura, la comprensió del codi i esclarirà per què s'està utilitzant la variable. Per això, seguirem les següents normes:

a. Nom de les variables autoexplicatiu.

El nom que li donem a una variable ha de representar l'element o els elements que contindrà. No ha de ser massa llarg, però tampoc massa abreujat. Haurem de trobar un punt mig que satisfaci les nostres necessitats.

Exemples de males declaracions:

```
int n;                              // Nom de la variable massa curt.
int comptador_de_preus_de_lector;   // Nom de la variable massa llarg.
int n_compt_cons;                   // Abreviació ambígua.
int dada;                           // Pot significar qualsevol cosa.
int cmptdr;                         // Elimina lletres.
int connexio_wgc;                   // Només tu saps qué és wgc.
```

Exemples de declaracions correctes:

```
int num_errors;                      // Num és una abreviació estàndard.
int connexio_dns;                    // DNS és un nom conegut.
int operand1;                        // Útil si existeix un "operand2".
char nom_fitxer[MAXF];               // Nom d'un fitxer.
```

Excepcions:

Es poden donar excepcions en el cas de variables puntuals de tipus auxiliar que només tenen sentit en una part molt reduïda del nostre codi.

```
int i, j, k;                         // Comptador d'un bucle.
int aux;                             // Auxiliar per fer un swap.
```

b. Nom de les variables en minúscula i separat per _.

Exemple de males declaracions:

```
int connexió_DNS;                    // No ha de contenir majúscules.
int numErrors;                       // Separar parules amb _.
int Dada;                            // No pot començar amb majúscula.
```

1.2. Declaració de variables

La ubicació i el format de la declaració de les variables també és molt important. Si les variables estan ben situades, podrem fer-nos una idea de la seva magnitud dins del procediment o funció.

a. *Variables declarades al principi*

Les variables hauran d'estar declarades al principi del procediment o funció i no enmig de sentències de codi.

Exemple de males declaracions:

```
int dia;
char lletra1;

dia = 1;
lletra1 = 'c';
int i;                                  //Variable mal declarada!
for (i = 0; i < 10; i++) {
      ...
```

La variable no s'ha declarat al principi i pot confondre més endavant (d'on ha sortit?).

b. *No utilitzar variables globals*

La utilització de variables globals de manera genèrica (sense ser per necessitat) és una molt mala pràctica en el món de la programació. Una variable global és confusa al no aclarir en quin moment exacte s'utilitza i al no poder-se saber quin valor té a cada moment.

c. *Valors de coma flotant*

Durant la declaració i utilització de valors de coma flotant s'ha d'utilitzar el nombre complet (amb coma) per evitar errors de comprensió i de programació.

Exemple de males declaracions:

```
float resultado = 5;            //resultado se debe declarar como 5.0.
int valor1 = 3;

resultat = valor1 / 2;          //Sin 2.0, la división es entera!
      ...
```

1.3. Consells

a. *Variables declarades en una sola línia*

Es recomanable que les variables que tenen relació entre sí estiguin declarades en una mateixa línia. L'objectiu és agrupar en blocs les variables que probablement s'utilitzin en el mateix moment.

Exemple:

```
int i, j, k;                                    //comptadors.
int day = 1, month = 1, year = 1900;            //variables de data.
char name[MAX_NAME], surname[MAX_SURNAME];      //variables de nom.
char letter_code;                               //altres variables
int num_users;
```

2. Constants

2.1. Nomenclatura de les constants

El nom que li donem a una constant és molt important. Facilita la lectura i comprensió del codi escrit. Per això, seguirem les següents indicacions:

a. Nom de les constants auto-explicatiu

El nom que li donem a una constant ha de representar el valor que contindrà. No ha de ser massa llarg, però tampoc massa breu. Haurem de trobar un punt mig que satisfaci les nostres necessitats.

Exemple de males declaracions:

```
#define N 10                            // Nom massa curt.
#define MAXIM_NUMERO_DE_USUARIS 100     // Nom massa llarg.
#define M_CON_EST 50                    // Abreviació ambígua.
#define MAX 10                          // Pot ser qualsevol cosa.
#define MX_CNXNS 50                     // Elimina letras.
#define MAX_WGC 6                       // Només tu saps què és wgc.
```

Exemple de declaracions correctes:

```
#define MAX_CONEXIONS 10                // MAX és una abreviació estàndard.
#define MY_IP “192.168.0.13”            // IP és un nom conegut.
#define MAX_ARRAY 100                   // Útil per definir arrays.
#define FILE_CONF “configuracio.txt”    // Nom d’un fitxer.
```

b. Nom de les constants en majúscula i separat per _.

Exemple de males declaracions:

```
#define max_CONEXIONS 10            // no pot contenir minúscules.
#define MAXCONEXIONS 10             // separar paraules amb _.
```

2.2. Declaració de constants

La ubicació de la declaració de les constants és molt important. Si no estan ben ubicades, la major part dels avantatges que aporten les constants es perden per no saber on s’han definit.

b. Constants entre llibreries i funcions

Las constants han de declarar-se just després dels *includes* de les llibreries que utilitzem al nostre programa o mòdul i, per tant, just abans de les declaracions de tipus.

Exemple de males declaracions:

```
#include <stdio.h>
#define MAX 10
typedef Enter int;
#include <stdlib.h>
#define N 15
```

Les constants estan repartides i barrejades entre llibreries i declaracions de funcions, de manera que es dificulta la seva lectura i ubicació.

Exemple de declaracions correctes:

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 10
#define N 15

typedef Enter int;
```

2.3. Consells

a. Definició de constants per cadenes i arrays

És recomanable que s'utilitzin constants per definir les mides de cadenes i arrays. Així, si en algun moment hem de redimensionar-los, només haurem de canviar el valor d'aquestes constants. En altres paraules, ajuda a la mantenibilitat del codi.

Exemple:

```
#define MAX_ARRAY 100

int main () {
	int i;
	int mi_array[MAX_ARRAY];

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

D'aquesta forma, si s'ha de canviar en algun moment la mida de l'array, només haurem de canviar el valor MAX_ARRAY, en comptes de canviar-lo en la declaració i la condició del *for*.

b. Definició de constants per valors amb significat

Si el nostre programa treballa amb valors que tenen un significat especial (per exemple, els colors d'una llibreria gràfica), definir constants per aquells valors sempre serà una bona pràctica. Sobretot si tenim en compte que probablement altres programes utilitzaran la nostra llibreria i no tenen per què conèixer els valors.

Exemple:

```
#define NEGRE      0
#define BLAU       1
#define VERD       2
#define CYAN       3
#define VERMELL    4
#define ROSA       5
#define TARONJA    6
#define GRIS_CLAR  7
 ...


int resetColor () {
      int color;
      color = NEGRE;                  // millor que “color = 0;”
      return color;
}
```

Així, sempre quedarà més clar assignar, comparar o fer qualsevol operació amb un color de valor NEGRE que 0.

3. Procediments i funcions

3.1. Nomenclatura dels procediments i funcions

El nom que li donem als nostres procediments i funcions és molt important. Facilita la lectura i comprensió del codi escrit. Per això, seguirem les següents indicacions:

a. Nom dels procediment i funcions auto-explicatiu

El nom dels procediments i funcions és especialment important, ja que encapsula un tros de codi amb sentit funcional. Una "funcionalitat" del nostre programa.

Exemple de males declaracions:

```
int f();                                // nom massa curt.
void sumarTotsElsElementsDelArray();    // nom massa llarg.
int contCon();                          // abreviacions ambigües.
int funcio();                           // pot significar qualsevol cosa.
void cntrCslls();                       // eliminar lletres.
int ghx2();                             // només tu saps què és ghx2.
char darthVader();                      // no té sentit
```

Exemple de declaracions correctes:

```
int trobarMax();                        // MAX és una abreviació estàndard.
void cambiarIP();                       // IP és un nom conegut.
int comptarElements();                  // està clar el que fa la funció.
void pintarMenu();
```

b. Nom de procediments i funcions en "lowerCamelCase"

"lowerCamelCase" és una convenció que consisteix a escriure diverses paraules sense espais entre elles. La primera lletra de cada paraula ha d'anar en majúscules excepte la primera lletra de la primera paraula. Un exemple d'això és "lowerCamelCase".

Exemple de males declaracions:

```
void lamevafuncio();            // falta majúscula a la m i a la f.
void LaMevaFuncio();            // la primera lletra ha de ser minúscula.
void LAMEVAFUNCIO();            // no pot anar tot en majúscules.
void la_meva_funcio();          // no ha d'haver-hi  _.
```

Exemple de declaració correcta:

```
void laMevaFuncio();            // compleix amb el format lowerCamelCase.
```

3.2. Mida dels procediments i funcions

Perquè l'ús de procediments i funcions sigui el correcte, haurem de delimitar la longitud:

c. *Mida màxima de procediments i funcions*

Un procediment o funció no ha de superar aproximadament les 60 línies de codi excepte excepcions per la pròpia naturalesa de la funció o procediment. El motiu és que, generalment, 60 línies de codi caben en tots els monitors i es podria llegir tot el codi sense moure el text.

Més enllà de la convenció estètica, la raó és que, si un procediment o funció és més llarg de 60 línies, en la majoria de casos serà perquè no haurem encapsulat bé la funcionalitat. Això voldrà dir que el nostre codi hauria "d'empaquetar-se" en més d'un procediment o funció.

4. Tipus

4.1. Nomenclatura dels tipus propis

El nom que li donem als nostres tipus propis és molt important. Facilita la lectura i comprensió del codi escrit. Per això, seguirem les següents indicacions:

a. Nom dels tipus auto-explicatiu

El nom dels tipus propis que definim ha de reflectir amb la màxima exactitud possible allò que representen. El nom ha de tenir una longitud adequada; ni massa llarg ni massa curt.

Exemple de males declaracions:

```
typedef struct {
      …
} T;                              // abreviació incomprensible i curta.

typedef struct {
      …
} ConjuntoDeDadesDeUsuari;     // massa llarg.

typedef struct {
      …
} Jug;                            // abreviació ambígua.
```

Exemple de declaracions correctes:

```
typedef struct {
      …
} Jugador;                        // defineix què conté perfectament.

typedef struct {
      …
} JuegoOca;
```

b. Nom dels tipus propis en "UpperCamelCase".

Aquesta convenció consisteix a escriure diverses paraules juntes sense separar-les per espais. Les primeres lletres de cada paraula aniran en majúscules sense excepció. Un exemple d'això és "UpperCamelCase".

Exemple de males declaracions:

```
typedef struct {
      ...
} ElMeutipus;                    // la t de “tipus” ha de ser majúscula.

typedef struct {
      ...
} elMeuTipus;                    // la primera lletra ha de ser majúscula.

typedef struct {
      ...
} ELMEUTIPUS;                    // no pot anar tot en majúscules.

typedef struct {
      ...
} el_meu_tipus;                  // no ha d’haver-hi  _.
```

Exemple de declaració correcta:

```
typedef struct {
      ...
} ElMeuTipus;                    // compleix la convenció UpperCamelCase.
```

5. Fitxers

5.1. Estructura d'un fitxer .c amb "main"

L'estructura d'un fitxer .c que contingui el procediment principal serà diferent de les altres, com veurem més endavant.

a. Llibreries

En primer lloc col·locarem les llibreries generals de sistema que utilitzem en ordre d'importància. Després col·locarem les llibreries específiques de sistema i, per últim, les llibreries que hàgim creat pel nostre projecte.

Exemple Includes:

```
// Llibreries generals de sistema
#include <stdio.h>
#include <stdlib.h>

// Llibreries específiques de sistema
#include <conio.h>
#include <Windows.h>

// Llibreries pròpies
#include "my_library.h"
```

b. Constants

Immediatament després de les llibreres, haurem de posar les constants que utilitzem en el nostre procediment principal que no estiguin en altres llibreries.

c. Tipus propis

En tercer lloc definirem els tipus propis. Tot i així, és més recomanable que els tipus propis estiguin en mòduls específics on es declarin i defineixin les operacions que podem realitzar amb ells.

d. Procediment principal

Finalment, escriurem el procediment principal. Ha de ser l'únic del fitxer i ha de ser suficientment breu i clar per permetre que es pugui entendre la globalitat del programa veient només aquest procediment...

Exemple de fitxer .c amb "main":

```
// Llibreries del sistema
#include <stdio.h>
#include <stdlib.h>

// Llibreries del nostre sistema
#include <conio.h>
#include <Windows.h>

// Llibreries pròpies
#include "la_meva_llibreria.h"

// Constants
#define CONSTANT1  1
#define CONSTANT2  2

// Tipus propis
typedef struct {
	int camp1;
	char camp2;
} ElMeuTipus;

// Procediment principal
int main (int argc, char* argv[]) {
	...
}
```

5.2. Estructura d'un fitxer .c de mòdul

L'estructura d'un fitxer .c que forma part d'un mòdul (conjunt de fitxers .c i .h) ha de ser la següent:

a. *Include del fitxer .h*

El primer que hauríem de trobar en un fitxer .c pertanyent a un mòdul és un include del seu fitxer .h corresponent. No afegirem més includes que aquest. Els includes que necessitem els farà el fitxer .h.

b. *Procediments i funcions*

A continuació haurà de trobar-se la implementació de tots els procediments i funcions del mòdul. En aquest fitxer no hauríem de trobar constants ni tipus, ja que aquests seran al fitxer.h del mòdul.

Exemple de fitxer .c:

```
// Include del fitxer .h del mateix mòdul
#include "el_meu_modul.h"

// Procediments i funcions
int funcio1 () {
      ....
}

int funcio2 () {
      ....
}

int funcio3 () {
      ....
}
```

5.3. Estructura d'un fitxer .h

L'estructura d'un fitxer .h serà la següent:

a. *"Define Guards"*

El primer que escriurem seran els "define guards". Són una eina a escale de compilador que evita que aquest entri en bucle de compilació i d'altres problemes. Per això, el primer que ha d'haver-hi en el nostre fitxer .h és el següent:

```
// Define Guard
#ifndef _EL_MEU_MODUL_H_
#define _EL_MEU_MODUL_H_

// Codi

#endif
```

_EL_MEU_MODUL_H_ serà el nom del nostre fitxer .h començant sempre pel caràcter _, escrit en majúscules i acabat en _. Això és així perquè estem definint una constant i les constants van en majúscules.

Haurem de tancar el fitxer amb la sentència #endif per tancar el condicional a escala de compilador.

b. *Llibreries*

En segon lloc estaran les llibreries que necessiti el nostre mòdul. Seguiran el mateix ordre que s'ha descrit en l'apartat b del punt 5.1. És a dir, primer les llibreries generals de sistema, després les llibreries específiques del sistema i, per últim les llibreries pròpies que pugui usar el mòdul.

Exemples d'includes:

```
// Llibreries generals de sistema
#include <stdio.h>
#include <stdlib.h>

// Llibreries específiques del sistema
#include <conio.h>
#include <Windows.h>

// Llibreries pròpies
#include "my_library.h"
```

c. *Constants*

Immediatament després de les llibreries, haurem de situar les constants que utilitzem en el nostre procediment principal que no estiguin en altres llibreries.

d. *Tipus propis*

A continuació definirem els tipus propis.

e. *Capçaleres de procediments i funcions*

Finalment, escriurem les capçaleres de tots els procediments i funcions que implementi el mòdul.

Exemple de fitxer .h:

```
// Define Guards
#ifndef  _EL_MEU_MODUL_H_
#define  _EL_MEU_MODUL_H_

// Llibreries del sistema
#include <stdio.h>
#include <stdlib.h>

// Llibreries del nostre sistema
#include <conio.h>
#include <Windows.h>

// Llibreries pròpies
#include "la_meva_llibreria.h"

// Constants
#define CONSTANT1  1
#define CONSTANT2  2

// Tipus propis
typedef struct {
      int camp1;
      char camp2;
} ElMeuTipus;

// Capçaleres dels procediments i funcions
int funcio1 ();
int funcio2 ();
int funcio3 ();

#endif
```

5.4. Consells

a. *Nomenclatura dels fitxers*

Es recomana seguir un format segons les regles:

- Els noms dels fitxers, tant els .c com els .h s'han d'escriure en minúscules i separats per '_'.
- El fitxer que contingui el procediment principal s'haurà d'anomenar main.c per facilitar la seva distinció respecte als .c dels altres mòduls.

b. *Fitxers .h y llibreries*

Si un fitxer .h inclou un altre que ja inclou les llibreries que necessita el primer, no fa falta que les inclogui. Tot i així, és una bona pràctica que s'incloguin.

Exemple:

Si el fitxer A.h inclou el mòdul B.h i el C.h, però el fitxer B.h ja inclou el C.h, A.h només necessitaria incloure a B.h per poder funcionar correctament:

```
#include “B.h”
```

Això és així perquè A.h inclou C.h a través de B.h.

Igualment, seria una bona pràctica que A.h inclogués ambdós fitxers .h per separat:

```
#include “B.h”
#include “C.h”
```

La raó és que facilitarà la comprensió, ja que el lector del mòdul sabrà exactament quins mòduls utilitza A.h sense haver d’entrar en el detall del codi escrit al fitxer A.c i buscar quines funcions utilitza de B.h i quines de C.h.

6. Espaiat

Per fer el nostre codi llegible i agradable a la vista, haurem de definir un espaiat comú que faciliti entendre l'estructuració de les sentències i el codi en general.

6.1. Espaiat vertical

L'espaiat vertical és el conjunt de línies en blanc que deixarem per fer el codi més visual.

a. *Espaiat de "paràgrafs"*

En tot codi de programació, com en l'escrit, es poden distingir diferents paràgrafs. Els paràgrafs són conjunts de línies que tenen sentit propi, cohesió. Així, en el nostre context, els paràgrafs seran conjunts de sentències que tendim a agrupar: definició de variables, bucles... Per fer més llegible el codi, deixarem una línia en blanc d'espai entre paràgrafs.

Exemple d'espaiat vertical:

```
#include <stdio.h>
#include <stdlib.h>

#define CONSTANT  1

int main () {
	int enter1;
	int enter2;
	char caracter1;

	for (enter1 = 0; enter1 < enter2; enter1++) {
		....
	}

	if ('3' == caracter1) {
		...
	}
}
```

6.2. Espaiat horitzontal

L'espaiat horitzontal és l'espaiat essencial que permetrà fer el codi llegible i jeràrquic.

a. *Operadors unaris*

Mai deixarem un espai en blanc per separar un operador unari de l'expressió amb la qual treballa.

Exemple de mal espaiat:

```
scanf(“%d”, & a);          // l’operador & i la variable han d’anar junts.
* p = 3;                   // l’operador * i la variable han d’anar junts.
i ++;                      // l’operador ++ i la variable han d’anar junts.
```

b. *Operadors binaris*

Sempre deixarem un espai per separar un operador binari de les expressions amb les quals treballa.

Exemple de mal espaiat:

```
a=b+c;                    // falten espais al voltant de = i +.
if (a==b) {               // falten espais al voltant de ==.
if (a<b) {                // falten espais al voltant de <.
```

Exemple correcte d’espaiat:

```
a = b + c;
if (a == b) {
if (a < b) {
```

Excepció: L’operador -> mai anirà espaiat.

c. *Comas i punts i comes*

Sempre deixarem un espai en blanc després d’una coma o punt i coma.

Exemple de mal espaiat:

```
int a,b,c;                    // falten espais després de les comes.
void func(int a,int b);       // faltan espais després de les comes.
for (a = 0;a < b;a++) {       // faltan espais després dels punt i coma.
```

d. *Parèntesis*

Sempre deixarem un espai en blanc fora dels parèntesis, tan d’obertura com de tancament. Però mai a l’interior.

Exemple de mal espaiat:

```
if(a == b){               // falten espais abans de ( i després de ).
if ( a == b ) {           // sobren espais després de ( i abans de ).
```

e. Claus

Sempre deixarem un espai abans de l'obertura de claus.

Exemple de mal espaiat:

```
if (a == b){                    // No hi ha espai abans de {
```

f. Switch-Case

Mai deixarem un espai abans dels ':' d'un "case" en una sentència switch.

Exemple de mal espaiat:

```
case 'A' :                      // no ha d'haver espai abans de :
```

7. Indentació

Perquè el nostre codi sigui llegible i agradable a la vista, haurem de definir un espaiat comú que faciliti entendre l'estructuració de les sentències i el codi en general.

7.1. Indentació de blocs

Un bloc comença amb una obertura de clau '{' i acaba quan aquesta clau es tanca. Per facilitar la lectura i estructuració del codi dins d'un bloc, el tabularem una vegada cap a la dreta. Les claus aniran a la mateixa alçada que la sentència que els ha obert.

En les sentències de switch, els "break" s'hauran de col·locar a la mateixa alçada que les sentències del bloc que conformen els "case".

Exemple de mala indentació:

```
#include <stdio.h>
#include <stdlib.h>

      #define CONSTANT  1          // no és un bloc nou.

int main () {
int enter1;                         // han d'indentar-se les variables.
int enter2;
char caracter1;

      if (condició) {
      sentencia1;                   // s'ha d'indentar la sentència.
                  }                 // la clau a l'alçada de l'if.

            switch (expressió) {    // no s'ha d'indentar. Està al nivell 1.
            case 1:
            sentencia2;             // s'ha d'indentar la sentència.
                  break;
            case 2:
                  sentencia3;
            break;                  // s'ha d'indentar el break.
      }
}
```

Exemple d'indentació correcta:

```
#include <stdio.h>
#include <stdlib.h>

#define CONSTANT  1

int main () {
	int enter1;
	int enter2;
	char caracter1;

	if (condició) {
		sentencia1;
	}

	switch (expressió) {
		case 1:
			sentencia2;
			break;
		case 2:
			sentencia3;
			break;
	}
}
```

7.2. Consells

És recomanable que cada indentació correspongui a una tabulació o al seu equivalent de quatre espais en blanc.

8. Comentaris

Els comentaris són una part essencial del nostre codi. Tot i que no afecten l'execució, faciliten enormement la comprensió del codi.

8.1. Comentaris de fitxers

Per entendre el sentit d'un mòdul i tenir suficient informació sobre ell, haurem d'afegir al principi del fitxer el propòsit del mòdul (per què s'ha escrit), els autors, la data de creació i la data de l'última modificació.

Capçalera de comentaris de fitxers:

```
/*********************************************
*
* @Propòsit:
* @Autor/s:
* @Data de creació:
* @Data de l'última modificació:
*
*********************************************/
```

8.2. Comentaris de funcions

Per facilitar la comprensió dels procediments i funcions que escriguem, afegirem sempre una capçalera explicativa. Haurem de definir la finalitat de la funció i explicar els paràmetres que rep com entrada/sortida i, si és una funció, què és el que retorna. Si no hi ha paràmetre de retorn, escriurem quatre guions ("----"):

Capçalera de comentaris de procediments i funcions:

```
/*********************************************
*
* @Finalitat:
* @Paràmetres:
* @Retorn:
*
*********************************************/
```

Exemple 1 de comentaris en procediments i funcions:

```
/*********************************************
*
* @Finalitat: Suma els valors dels paràmetres introduïts.
* @Paràmetres:     in: numero1 = primer operand de la suma.
*                  in: numero2 = segon operand de la suma.
* @Retorn: Retorna la suma de numero1 + numero2.
*
*********************************************/
int sumar (int numero1, int numero2);
```

Exemple 2 de comentaris en procediments i funcions:

```
/*********************************************
*
* @Finalitat: crea una cua amb fantasma i la deixa buida.
* @Paràmetres:     in/out: la_cua = cua no definida que passarà a guardar
                   la cua.
*                  in: fantasma = si val 0 es crearà sense fantasma. Per
                   qualsevol altre valor, es crearà amb fantasma.
* @Retorn: ----.
*
*********************************************/
void crearCua (Cua *la_cua, int fantasma);
```

8.3. Comentaris de línia

Per esclarir algunes parts del nostre codi, haurem d'afegir comentaris entre línies. Aquests comentaris han de servir per explicar parts complexes sobre com hem implementat quelcom en concret. Les explicacions han de ser concises i amb sentit; no haurem d'escriure ni obvietats ni explicacions ofuscades o poc clares.

Exemple de mal comentari:

```
// Incremento el valor de 'x' en 1.
x = x + 1;

// Faig un bucle fins MAX_ELEM.
for (i = 0; i < MAX_ELEM; i++) {
      if (max < arr[i]) {
            max = arr[i];
      }
}
```

Exemple de comentaris correctes:

```
// Bucle que busca el valor màxim en l'array arr.
for (i = 0; i < MAX_ELEM; i++) {
      if (max < arr[i]) {
            max = arr[i];
      }
}
```

a. Comentaris laterals

Si es decideix afegir comentaris en la mateixa línia de codi per clarificar la mateixa, és necessari que estiguin tots tabulats a la mateixa alçada.

Exemple de comentaris mal tabulats:

```
int numero;  // Guarda el valor del nombre introduït per l'usuari.
char nom_usuari[MAX]; // Nom de l'usuari que introdueix el valor.
int edat;    // Edat de l'usuari.
```

Exemple de comentaris ben tabulats:

```
int numero;               // Guarda el valor del nombre introduït per
                          // l'usuari.
char nom_usuari[MAX];     // Nom de l'usuari que introdueix el valor.
int edat;                 // Edat de l'usuario.
```

9. Control d'errors

Tot i que no afecten la comprensió del programa, un bon programador ha de gestionar i tenir un bon control dels possibles errors del seu programa.

9.1. Control

Tots els possibles errors hauran d'estar controlats pel programador de manera que si l'usuari comet un error treballant amb el programa, aquest no deixi de funcionar.

9.2. Missatges d'error

Tots els errors que afecten l'usuari (introducció de dades, escollir una opció inexistent...) hauran de mostrar un missatge d'error que avisi a l'usuari donant-li detalls de per què s'ha produït l'error. Els missatges han de ser breus però clars.

Mals missatges:

```
"Error"                          // no explica el motiu de l'error.
"No va"                          // ¿per què?
"Error, dades incorrectes"       // ¿quina dada és la que està malament?
```

Missatges correctes:

```
"Error, la dada introduïda no està dins del rang vàlid [1..15]"
```

10. Gestió de memòria

Tot i no afectar a la comprensió del programa, un bon programador sempre gestionarà i farà bon ús de la memòria del sistema amb la qual treballa.

10.1. Control

Sempre que es demani memòria de forma dinàmica haurem de comprovar que se'ns ha atorgat sense problemes. En cas d'error, mostrarem un missatge segons s'ha explicat a la "secció 9".

10.2. Gestió

Sempre que s'utilitzi memòria dinàmica haurem de comprovar que, al final del programa, quedi alliberada.