Guía de estilo para programar en C

José Fidel Argudo Argudo
Mª Teresa García Horcajadas
Nuria Hurtado Rodríguez
Depto. Lenguajes y Sistemas Informáticos
Escuela Superior de Ingeniería
Universidad de Cádiz

Índice

1. Identificadores

Los identificadores nombran elementos del programa que representan entidades del problema, por lo que tienen que ser **significativos** y proporcionar una breve, pero clara, descripción de las entidades a las que se refieren. El objetivo al elegir identificadores no es minimizar el número de pulsaciones, sino **contribuir**

a la legibilidad del programa. En general, conviene utilizar palabras completas o abreviaturas que se entiendan con facilidad, tratando de evitar secuencias de caracteres sin significado (a, b, xy, tbi,...) y los nombres con un significado muy amplio (variable, funcion, calcular, imprimir,...)

1.1. Constantes y macros

Se escribirán en mayúsculas, separando las palabras que compongan un identificador mediante el carácter de subrayado o guión bajo ('_').

```
#define FALSO 0
#define VERDADERO 1
#define DIAS_SEMANA 7
#define MAXIMO(x,y) (((x) >= (y)) ? (x) : (y))
```

1.2. Variables y parámetros

Como norma general se utilizarán letras minúsculas. En el caso de que el identificador esté formado por varias palabras, éstas se separarán con el carácter de subrayado ('_'), o bien se escribirá en mayúscula la letra inicial de cada una.

```
cont
distancia
VolumenEsfera
precio_total
```

Se podrán usar identificadores como `i`, `j` o `k` para las variables de control de bucles (`for`, `while`, `do...while`), siempre que el objetivo principal de la variable sea precisamente éste y no tenga otro significado especial. Excepcionalmente, dentro de una función o programa pequeño, se podrán usar estos nombres de variables, si no dificultan la comprensión, aunque no es recomendable.

1.3. Tipos de datos

Se escribirán igual que las variables (en minúsculas y separando las palabras con las iniciales en mayúscula o con el carácter '_'), pero utilizando el prefijo `t` o `tipo` para distinguirlos de aquellas.

```
tCadena
t_texto
tipoPalabra
tipo_tabla
```

1.4. Funciones

Se utilizarán, preferiblemente, verbos en infinitivo, sustantivos que indiquen acción o el nombre del resultado que devuelve la función y se escribirán en minúsculas, separando las palabras que compongan un identificador con las iniciales en mayúscula o con el carácter '_'.

```
ImprimirFichero()
BorrarCaracter()
escribir_opciones()
barajar()
intercambio()
grados_a_radianes()
MaximoComunDivisor()
```

2. Comentarios

En general, se puede decir que un programa escrito siguiendo un buen estilo, con un formato consistente y una estructura espacial adecuada, contribuirá en gran medida a su comprensión y estará autodocumentado. Los comentarios se emplearán para aumentar la claridad del programa, cuando la lógica del mismo conlleve cierta complejidad, pero no se debe abusar de ellos y nunca se utilizarán para compensar un código difícil de leer y comprender.

2.1. Estilo

Se distinguirán los siguientes tipos de comentarios:

- **Comentarios de bloque**: Se utilizarán para comentarios largos que ocupen más de una línea, justo delante del bloque que comentan alineados en la misma columna que éste.

```
/*
 * Un comentario de bloque
 * que ocupa dos líneas
 */
```

- **Comentarios enmarcados**: Son comentarios de bloque resaltados que se delimitan con dos líneas de guiones ('-') o asteriscos ('*').

```
/*
 *----------------------------------------
 * Un comentario enmarcado
 * que ocupa dos líneas
 *----------------------------------------
 */
```

- **Comentarios cortos**: Se escribirán en la línea que precede al bloque de sentencias que comentan alineados en la misma columna que éste.

```
/* Comentario corto (una sola línea) */
```

- **Comentarios de fin de línea**: Comentan sólo una línea de código y se separarán de ésta con varios espacios.

```
int i;   /* Comentario de fin de línea */
```

2.2. Contenido

- **Comentarios cortos y de bloque**: Estos se emplearán para documentar bloques de sentencias o las funciones cuyo nombre no explique suficientemente su cometido. Hay que explicar *qué* hace el código comentado y evitar describir *cómo* funciona.

  Si el código es farragoso y poco legible, en lugar de tratar de aclararlo haciendo uso de comentarios, se debería reescribir para hacerlo más entendible.

- **Comentarios enmarcados**: Se escribirán al principio del programa (comentario inicial) o de cada fichero del mismo para permitir seguir un poco su historia, indicando: Nombre del programa, objetivo, parámetros (si los tiene), condiciones de ejecución, módulos que lo componen, autor o autores, fecha de finalización, últimas modificaciones realizadas y sus fechas y cualquier otra eventualidad de la que el programador quiera dejar constancia.

- **Comentarios de fin de línea**: Se emplearán en las declaraciones de variables y en las definiciones de constantes y tipos de datos cuando sus nombres no indiquen claramente su significado y utilidad. También se utilizarán al cerrar un bloque con '}', para indicar a qué sentencia de control de flujo pertenece, principalmente cuando existan diversas sentencias anidadas o el bloque sea muy largo.

No debemos olvidar que los comentarios son textos literarios que probablemente leerán otras personas, por lo que debemos cuidar el estilo, la sintaxis y la ortografía.

3. Formato del programa

El lenguaje C es muy poco restrictivo en cuanto al formato de los programas. Podemos escribir cada instrucción con los espacios que queramos, incluso ninguno, entre los elementos o *tokens* que la componen. También podemos escribir un programa completo en una sola línea o separar las instrucciones con tantas líneas en blanco como queramos. Sin embargo, es muy importante adoptar un formato que facilite la lectura y ayude a la comprensión del código.

3.1. Estructura del código del programa

Para estructurar adecuadamente el código de un fichero o módulo de un programa en C se seguirá la siguiente organización (fig. 1):

1. Comentario inicial: Un comentario enmarcado describiendo el fichero.

2. Bibliotecas del sistema: Los ficheros `.h` con el `#include` y entre ángulos (`<...>`) el nombre del fichero de cabecera. Quizás la más típica sea:

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

3. Bibliotecas propias de la aplicación. Normalmente, en grandes aplicaciones, se suelen realizar varias bibliotecas con funciones, separadas por su semántica. Los nombres de fichero se ponen entre comillas (para que el compilador no las busque en el directorio de las bibliotecas estándar) y se puede incluir un comentario aclaratorio:

```
#include "rata.h"  /* Rutinas para control del ratón */
#include "cola.h"  /* Funciones para manejar colas   */
```

4. Variables globales con la palabra reservada `extern`, es decir, usadas en el módulo pero declaradas en otro módulo distinto.

5. Constantes simbólicas y definiciones de macros, con `#define`.

6. Definición de tipos, con `typedef`.

7. Declaración de funciones del módulo: Se escribirá sólo el prototipo o cabecera de la función, no su definición o implementación. De esta forma, el compilador (y el programador) conocerá el número de parámetros y el tipo de cada uno, así como el tipo del valor de retorno.

8. Declaración de variables globales del módulo: Se trata de las variables globales declaradas y usadas en el propio módulo.

9. Implementación de funciones: Aquí se programarán las acciones de cada función, incluida la función principal `main()`. Si el módulo incluye la función `main()`, ésta se sitúa la primera y el resto de funciones, se pueden ordenar por orden de aparición de la llamada en la función `main()` y las funciones que son llamadas desde otras conviene escribirlas consecutivas para facilitar la lectura. Es una buena medida, que los prototipos de las funciones aparezcan en el mismo orden que sus definiciones, ya que así puede ser más fácil localizarlas.

3.2. Espacios en blanco

- Cuando haya que escribir una coma, ésta debe ir junto a la palabra que le precede y seguida de un espacio en blanco.
- Se debe incluir un espacio en blanco antes y después de un operador de asignación (=, +=, -=, *=, /=, %=) y de cada operador binario aritmético (+, -, *, /, %), lógico (`&&, ||`) o relacional (`==, !=, <, >, <=, >=`).
- El paréntesis '(' va inmediatamente seguido de la siguiente palabra sin espacio en medio. El paréntesis ')' va inmediatamente precedido de la palabra anterior sin espacio entremedias.

3.3. Fin de línea y líneas en blanco

- Cada instrucción se escribirá en una línea, salvo que sea demasiado larga. En este caso, se dividirá en varias líneas sangradas respecto a la primera, pero eligiendo los puntos de división de tal forma que no dificulten la compresión de la instrucción.

```
posicion[i] = (Espacio[Espacio[i+1].sigte].sigte + 1) *
              Espacio[Espacio[i].sigte].sigte;

fseek(fsecind->fichero,
      fsecind->indice[i].pos*sizeof(registro),
      SEEK_SET);

if ((expresion1 || expresion2) &&
    (expresion3 || expresion4)) {
    ....
}

printf("%s\n", "Esta es una cadena de caracteres "
       "muy larga y por eso la dividimos en "
       "varias líneas.");
```

- Separaremos el final de la implementación de una función y el inicio de la siguiente con al menos una línea en blanco.
- Las diferentes partes de un programa (ficheros de cabecera, definiciones de constantes y tipos, prototipos de funciones, declaraciones de variables,...) deben separarse también con una o más líneas en blanco.

3.4. Sangrado

El sangrado deberá ser consistente en todo el programa, es decir, siempre utilizaremos el mismo número de espacios para cada nivel de sangrado. Utilizar un sangrado de dos espacios puede ser suficiente para apreciar la estructura del programa, pero no conviene utilizar más de cuatro espacios para evitar un excesivo desplazamiento horizontal de las líneas, sobre todo cuando se escriben varias estructuras de control anidadas.

3.5. Declaración de variables

Al declarar un conjunto de variables las agruparemos por tipo, declarando todas las del mismo tipo en la misma línea, si no cupieran se dividirán en varias líneas sangradas respecto a la primera variable de ese tipo. Las variables que necesiten un comentario explicativo, se declaran cada una en una línea con un comentario de fin de línea.

```
int a, b, c,
    n,   /* número total de valores */
    media;   /* valor medio de n valores */
char *nombre, *apellidos, *direccion,
     *poblacion;
```

3.6. Definición de estructuras

Una estructura o registro se define mediante la palabra reservada `struct` seguida del nombre de la estructura y, encerrada entre llaves ('{' y '}'), la lista

de campos que la componen separados por comas. Cada campo se escribe en una línea sangrado respecto a la palabra `struct`, indicando su tipo y nombre del campo. La llave de inicio ('{') se coloca a continuación del nombre de la estructura en la misma línea y la que finaliza la definición ('}') después del último campo alineada con la palabra `struct`.

```
struct empleado {
   char nif[9],
   char nombre[16],
   char apellidos[32],
   int cod_depto,   /* código departamento [0,10] */
   int cod_cat,   /* código categoría [0,5] */
   char puesto[16],
   float sueldo,
};
```

3.7. Formato de instrucciones de control del flujo

El bloque de instrucciones asociado a una estructura de control del flujo se escribirá debajo y sangrado respecto a ésta. Además se encerrará entre llaves ('{' y '}') sólo cuando conste de más de una sentencia, excepto cuando tratándose de una única instrucción la lógica del programa obligue a escribirla entre llaves, como en el caso de estructuras selectivas anidadas.

```
if (a > N)
{
   if (a < M)
      a = x + 1;
}
else
{
   a = x - 1;
   z = a;
}
```

La llave de inicio del bloque ('{') asociado a una estructura de control se puede colocar en la línea inferior alineada en la misma columna que dicha estructura, mientras que la llave de fin de bloque ('}') se colocará en la línea siguiente a la última instrucción sola y alineada en la misma columna que la de apertura. De esta forma es fácil descubrir cuáles son las llaves que se emparejan y detectar los errores.

Una alternativa consiste en colocar la llave de inicio de un bloque en la misma línea que la estructura de control a la que va asociado y escribir la llave de fin de bloque alineada con dicha estructura. Esto proporciona menos líneas de código sin pérdida de claridad, ya que el sangrado ayuda a identificar claramente las instrucciones que componen el bloque.

3.7.1. Estructuras selectivas

La estructura selectiva se escribe alineando en la misma columna las palabras reservadas `if` y `else` y sangrando sus respectivos bloques de instrucciones.

```
if (expresion)
{
   instruccion1;
   instruccion2;
   ...
}
else
{
   instruccion1;
   instruccion2;
   ...
}
```

```
if (expresion) {
   instruccion1;
   instruccion2;
   ...
}
else {
   instruccion1;
   instruccion2;
   ...
}
```

Las estructuras selectivas anidadas, cuando la sentencias `else` consisten cada una en un solo `if` (cascada `else-if`), se pueden escribir alineando todas las palabras `else` en la misma columna que el primer `if` y cada una con el siguiente `if` en la misma línea, de esta forma se evita el excesivo desplazamiento del código hacia la derecha.

```
if (expresion1)
{
   instruccion1;
   instruccion2;
   ...
}
else if (expresion2)
{
   instruccion1;
   instruccion2;
   ...
}
else if (expresion3)
{
   instruccion1;
   instruccion2;
   ...
}
else
{
   instruccion1;
   instruccion2;
   ...
}
```

```
if (expresion1) {
   instruccion1;
   instruccion2;
   ...
}
else if (expresion2) {
   instruccion1;
   instruccion2;
   ...
}
else if (expresion3) {
   instruccion1;
   instruccion2;
   ...
}
else {
   instruccion1;
   instruccion2;
   ...
}
```

La anterior es una estructura selectiva múltiple que equivale a una estructura `switch`. En ésta las palabras `case` van sangradas respecto a `switch` y alineadas en la misma columna. El bloque de instrucciones de cada caso se escribe sangrado respecto a la palabra `case`.

```
switch (selector)
{
   case valor1:
      instruccion11;
      instruccion12;
      ...
      break;
   case valor2:
   case valor3:
      instruccion21;
      instruccion22;
      ...
      break;
   ...
   case valorN:
      instruccionN1;
      instruccionN2;
      ...
      break;
   default:
      instruccion1;
      instruccion2;
      ...
}
```

```
switch (selector) {
   case valor1:
      instruccion11;
      instruccion12;
      ...
      break;
   case valor2:
   case valor3:
      instruccion21;
      instruccion22;
      ...
      break;
   ...
   case valorN:
      instruccionN1;
      instruccionN2;
      ...
      break;
   default:
      instruccion1;
      instruccion2;
      ...
}
```

3.7.2. Estructuras repetitivas

En los bucles `for` se escribirán las tres expresiones de control en la misma línea que la palabra `for`, entre paréntesis y separadas por un punto y coma seguido de un espacio. Si la línea es demasiado larga se seguirán las normas del apartado 3.3 para dividirla en varias líneas, tratando de terminar cada una en un punto y coma para que cada expresión aparezca completa en una línea. En cualquier caso, la instrucción o bloque de instrucciones del bucle se escribirá sangrado respecto a éste.

```
for (expresion1; expresion2; expresion3)
{
   instruccion1;
   instruccion2;
   ...
}
```

```
for (expresion1;
     expresion2_muy_larga;
     expresion3)
{
   instruccion1;
   instruccion2;
```

```
   ...
}


for (expresion1; expresion2; expresion3) {
   instruccion1;
   instruccion2;
   ...
}
```

La expresión de un bucle `while` se escribirá en la misma línea que la palabra `while` y el bloque de instrucciones en las líneas siguientes sangrado respecto al bucle.

```
while (expresion)
{
   instruccion1;
   instruccion2;
   ...
}
```

```
while (expresion) {
   instruccion1;
   instruccion2;
   ...
}
```

En un bucle `do-while` el bloque de instrucciones se escribe sangrado debajo de la palabra `do`. La palabra `while` se escribe en la misma línea que la llave de cierre del bloque de instrucciones separada por un espacio.

```
do
{
   instruccion1;
   instruccion2;
   ...
} while (expresion);
```

```
do {
   instruccion1;
   instruccion2;
   ...
} while (expresion);
```

3.8. Funciones

3.8.1. Declaración

Antes de utilizar una función en cualquier programa es conveniente declararla, ya que si no se hace así cualquier persona que lea el código no tendrá ninguna información sobre la función y le resultará difícil entenderlo. Además si una función no está declarada antes de ser llamada, pueden producirse errores de compilación y también de ejecución. La declaración de una función se hace escribiendo su cabecera o prototipo. En el apartado 3.1 se indica el lugar adecuado para declarar las funciones de un módulo.

El prototipo de una función se escribirá en una sola línea indicando, opcionalmente, el nombre de los parámetros. Si no caben todos los parámetros en la misma línea, se escribirán en varias alineados en la misma columna que el primer parámetro.

```
void funcion (int, int, double, char *);

void funcion (int i, int j, double z, char *s);

void funcion (int parametro_entero1, int parametro_entero2,
              double parametro_double, char *cadena);
```

3.8.2. Definición

La definición de una función se compone de los siguientes elementos:

- Comentario de bloque explicando su funcionalidad (si fuera necesario), que contiene información sobre la especificación de la función (precondiciones y postcondiciones).
- Cabecera alineada en la primera columna.
- Declaración de variables e instrucciones encerradas entre llaves ('{' y '}') y sangradas respecto a la cabecera. La llave de apertura se escribe debajo de la cabecera en la primera columna y la llave de cierre después de la última instrucción, también en la primera columna.

4. Otras normas

- No utilizar `break` (salvo en la sentencia `switch`) ni `continue`.
- Cada función debe tener un solo punto de salida, un solo `return`.
- No usar variables globales para evitar efectos laterales.
- No declarar e inicializar variables en la misma línea.

```
/*
  *-----------------------------------------------------
  * ejemplo.c
  * Autor: Juan Garcia Perez
  * Fecha: 15-07-2004
  * Descripcion: Breve descripcion de la funcionalidad
  * de este modulo.
  *-----------------------------------------------------
  */

#include <stdio.h>
#include <stdlib.h>
#include "figuras.h"
#include "colores.h"

extern double superficie;
extern int n_lados;

#define PI 3.141592
#define PUL_CM(x) ((x) * 2.54)
#define CM_PUL(x) ((x) / 2.54)

typedef enum {FALSO, VERDADERO} t_boolean;
typedef char t_cadena[32];
typedef struct {
           t_figura *fig;
           int n_fig;
        } t_lista_figuras;

int colorear (t_figura fig, t_color color);
int pintar (t_figura fig, int escala);

int n_figuras;
double superficie_total, perimetro;

int main ()
{
  ...
}

int colorear (t_figura fig, t_color color)
{
  ...
}

int pintar (t_figura fig, int escala)
{
  ...
}
```

1. Comentario inicial

2. Cabeceras de la biblioteca estándar

3. Cabeceras propias de la aplicación

4. Variables globales de otros módulos

5. Constantes simbólicas y macros

6. Definición de tipos

7. Declaración de funciones

8. Variables globales del módulo

9. Implementación de funciones

Figura 1: Estructura de un módulo de un programa en C.