Consejos y normas de estilo para programar en C

Tabla de Contenidos

1. Introducción

Se ha insistido a lo largo del curso en la importancia de programar de forma estructurada (utilizando las estructuras de control vistas en clase), y de la utilidad de hacerlo de forma modular (módulos o subprogramas reutilizables en los que resulta más fácil la detección de errores en cada módulo que en el total del programa si no tuviera módulos).

Al mismo tiempo se hace hincapié en que el código de nuestros programas debe ser legible. Esto implica claridad en su escritura, para facilitar su posterior lectura, mantenimiento y modificación. Es también importante el uso de comentarios que permitan el entendimiento del programa a personas ajenas al mismo o al propio programador una vez pasado un cierto tiempo.

Para ello intentaremos aplicar una serie de normas de "estilo" a la hora de programar. Aquí daremos las principales, y a lo largo de las prácticas iremos insistiendo en ellas y añadiremos algunas otras.

2. El fichero fuente

Existe una estructura global que se debe respetar, que será la siguiente:

- Todos los ficheros deben comenzar con un comentario, con el nombre del fichero, autor, fecha de realización, y una breve descripción de su contenido. Existen numerosos estilos para este tipo de introducciones. Se recomienda que no sea demasiado extenso.
- A continuación se deben incluir los ficheros de cabecera (ficheros `.h`), mediante las directivas de preprocesado `#include`. Primero los ficheros de cabecera del sistema (`stdio.h`, `stdlib.h`, etc.) y luego los propios de la aplicación.
- Constantes simbólicas y tipos definidos por el usuario.
- Variables globales si las hubiera (en esta asignatura no están permitidas, ya que pueden da lugar a errores difíciles de detectar).
- Prototipos de funciones locales al fichero (que no se usan en otros ficheros).
- Funciones: primero `main` (si existe), y a continuación el resto, agrupadas por algún criterio. Si no existe criterio, al menos alfabéticamente. Se recuerda que la declararción o prototipo de una función debe aparecer antes de ser llamada.

Cada una de estas secciones debe estar separada de las demás por una línea en blanco, y tener un comentario que la preceda:

```
/*
**    Fichero:   demostraciones.c
**    Autor:     Fulanito Menganito Blas
**    Fecha:     21-12-99
**
**    Descripción:   conjunto de funciones de
**                   demostraciones matemáticas.
**
**
*/

/* Includes del sistema */
#include <stdio.h>

/* Includes de la aplicación */
#include "ejemplos.h"

/* constantes */
#define MAXIMO   100
#define ERROR    "Opción no permitida"
/* tipos definidos por el usuario */

/* Prototipo de funciones locales */

int main()
{

  return 0;
}

/* Definiciones de funciones locales */
```

3. Sangrado y separaciones

Se debe usar el espaciado tanto vertical como horizontal "generosamente", para reflejar la estructura de bloques del código (a esto se le llama sangrado o indentación). Un programa bien indentado será mucho más legible que uno mal indentado o sin indentar.

Sólo debemos tener una sentencia por línea:

```
if (correcto)
  printf("no hubo problemas");
else
  printf("SI hubo problemas");
```

en vez de

```
if (correcto) printf("no hubo problemas");
else printf("SI hubo problemas");
```

Dentro de una función los distintos bloques deben ir separados por líneas en blanco:

```
#include <stdio.h>

int main()
{
  int num=0;          /* número a leer de teclado */
  int contador=0;     /* contador total */
  int impares=0;      /* contador de impares */
  int positivos=0;    /* contador de positivos */
  int resp;           /* respuesta del usuario */

  /* Separamos definiciones de código */
  do
    {
      printf("\nIntroduzca un número: ");
      scanf("%d", &num);
      contador++;
      if (num>0)
        positivos++;
      if (num%2)
        impares++;
      while (getchar()!='\n')   /* leer hasta  */
        ;                       /* nueva_linea */
      printf("Terminar (S/N)?");
      resp = getchar();
    }
 while (resp!='S');
  printf("Numeros: %d\t Pares: %d\t Positivos: %d\n",
         contador, contador - impares, positivos);
  return 0;
}
```

Hay distintos estilos a la hora de utilizar las llaves de bloques, bucles, etc. Se recomienda utilizar siempre el mismo. El propuesto es:

```
control
 {
  sentencia;
  sentencia;
 }
```

ya que es visualmente sencillo comprobar los pares de llaves. El propuesto por K&R es:

```
control {
  sentencia;
  sentencia;
}
```

Cuando tengamos una expresión condicional de bastante longitud, se debe separar en varias líneas, tabulando el comienzo de cada una para que queden alineadas las condiciones:

```
/* Incorrecto: ¡No demasiado legible! */
while ( cond1 != "error"  && cond2 !=TRUE &&   cond3 >=
7.5 && cond4 <=10 && cond5 !="esto es verdad")
```

Queda mucho más claro si se pone:

```
/* correcto */
while (cond1 != "error"  && cond2 != TRUE &&
       cond3 >= 7.5 && cond4 <= 10 &&
       cond5 != "esto es verdad")
```

Debido a que es posible que nuestro fichero se utilice en distintas plataformas, y se edite con distintos editores de texto, se debe limitar la longitud de las líneas a 70 caracteres. Por ejemplo, si estamos realizando comentarios al código y sobrepasamos esta longitud se debe introducir un retorno de carro y continuar en la siguiente línea.

Líneas de gran longitud debidas a una tabulación demasiado profunda a veces reflejan una pobre organización del código.

4. Comentarios

> *Cuando el código y los comentarios no coinciden, probablemente ambos sean incorrectos. (Norm Schryer).*

Los comentarios breves se pueden realizar a la derecha del código, y con una cierta tabulación para que no se confunda con el código. Se debe procurar que todos los comentarios estén al mismo nivel.

Cuando se comenta un bloque de código se puede comentar previamente y al mismo nivel de sangrado.

```
/*
** bucle que mientras el depósito no está lleno aumenta
** el caudal siguiendo las tablas de Leibniz-Kirchoff
*/
while(nolleno)
  {
     ?
  }
```

Tan malo es no comentar como comentar en exceso o innecesariamente. Hay que evitar comentarios que se puedan extraer o deducir fácilmente del código:

```
i++;    /* Se autoincrementa el valor de i en una unidad */
```

Cuando se define una función se debe preceder de un comentario con el funcionamiento básico de la función.

Se deben comentar brevemente los prototipos de las funciones en los ficheros de cabecera, explicando el significado de los parámetros y el valor devuelto por la función. Los detalles del funcionamiento interno estarían en los comentarios de la definición de la función.

5. Identificadores y Constantes

Se deben evitar los identificadores que comiencen o terminen con el carácter de subrayado (_), ya que suelen estar reservados para el sistema.

Recuerde que los identificadores deben ser representativos del valor que identifican. Además le ahorrarán comentarios:

```
velocidad = espacio/tiempo;
```

en vez de

```
resultado = valor/aux2;
```

Si se usan variables locales en distintas funciones, pero con el mismo significado, ayuda a la legibilidad mantener el identificador. En cambio, confunde utilizar el mismo identificador con diferentes significados.

A la hora de definir variables, se debe definir una por línea. Al mismo tiempo los tipos, identificadores y comentarios deben aparecer al mismo nivel para dar mayor claridad y facilidad de lectura:

```
int trayecto[100];   /* comentario sobre trayecto */
int origen;          /* comentario sobre origen   */
int destino;         /* comentario sobre destino  */
```

Evite identificadores que difieran en una sola letra de otros, la confusión es muy probable.

Evite el uso de identificadores que puedan coincidir con bibliotecas del sistema.

Para evitar la confusión entre asignaciones y comparaciones, es conveniente en una primera época utilizar en las comparaciones la constante a la izquierda de la variable para obtener un error en compilación si realizamos una asignación (=) en lugar de una comparación (==).

```
if ( 4 == a )
```

en lugar de:

```
if ( a == 4 )
```

Las constantes simbólicas deben ir en MAYÚSCULAS.

Las funciones, nombres de variables, tipos definidos por el usuario, etiquetas de estructuras y uniones irán en minúsculas.

Las constantes de una enumeración van todas en mayúsculas, o al menos con la primera letra en mayúsculas.

En general, los valores numéricos no deben ser usados directamente en el código, dan lugar a lo que se conoce como ``números mágicos'', que hacen el código difícil de mantener y de leer. Para ello se usan las constantes simbólicas, o las enumeraciones.

Las constantes de tipo carácter deben ser definidas como literales y no como sus equivalentes numéricos:

```
#define CERO    48      /* INCORRECTO */
#define CERO    '0'     /* CORRECTO */
#define A       'A'
```

El calificador de puntero * debe ir con el nombre de la variable en vez de con el tipo.

```
int   *pi;
```

y no se deben mezclar de la forma

```
int   x, y, *z;
```

En una estructura o unión, los elementos de ésta deben ir cada uno en una línea, tabulados y con un comentario:

```
struct barco
  {
    int flotacion;   /* en metros */
    int tipo;     .  /* según clasif. internacional */
    float precio;    /* en dólares */
  };
```

6. Programación Modular y Estructurada

Sólo se deben utilizar las estructuras que la programación estructurada recomienda.

No se deben usar las sentencias **`goto`** o **`continue`**:

> *Su uso [el de la sentencia goto] está restringido en una programación estructurada, pues tiende a generar programas ilegibles de difícil entendimiento y comprensión, lo que dificulta, y en ocasiones impide, cualquier tipo de actualización o modificación sobre los mismos, obligando al programador a desarrollar un nuevo programa o aplicación. (Fundamentos de Programación, J. López Herranz, E. Quero Catalina, Ed. Paraninfo).*
>
> *In the case of the goto statement, it has long been observed that unfettered use of goto's quickly leads to unmaintainable spaghetti code. (Indian Hill Recommended C Style and Coding Standards).*

La sentencia **`break`** sólo tiene justificado su uso en la selectiva múltiple, es decir, con la sentencia compuesta **`switch`**.

Toda función tendrá un único punto de retorno (una única sentencia **`return`**), que además deberá ser la última línea de la función. Además, deberá devolver el control a la función que la invocó, y no a otra o al sistema.

Aunque no existen reglas estrictas, el código de una función no debería sobrepasar una o dos páginas. Si no es así, posiblemente no se está descomponiendo el programa adecuadamente.

Siempre se debe especificar el tipo de retorno de una función. Si no lo tiene, especificar **`void`**.

La división en funciones se debe hacer de forma que cada función tenga una tarea bien definida. Al menos deben codificarse en forma de función aquellas tareas con cierta entidad que se realizan dos o más veces en un programa.

7. Sentencias Compuestas

Las sentencias compuestas son aquéllas que van encerradas entre llaves. Hay varias formas de formatear estas llaves, pero es importante seguir siempre una. La propuesta, como ya vimos en un apartado anterior es:

```
control
  {
     sentencia;
     sentencia;
  }
```

Si el cuerpo de un bucle o función es nulo debe estar sólo en una línea, y comentado para saber que no es un error u olvido, sino intencionado.

El cuerpo de los bucles **`do-while`** siempre debe ir entre llaves.

7.1 switch

Las etiquetas deben ir en líneas diferentes, y de igual modo el código asociado a cada una de ellas.

Se debe incluir siempre sentencia **`break`**, y si se desea un comportamiento "en cascada" se debe comentar muy claramente:

```
switch (expresion_entera)
{
    case ETIQ_1:        /* igual que ETIQ_2 */
    case ETIQ_2:
      sentencia;
      break;
    case ETIQ_3:
      sentencia;
      break;
}
```

7.2 if-else

Si cualquiera de los brazos tiene una sentencia compuesta, se recomienda que ambos brazos vayan encerrados entre llaves. Estos además son necesarios en el caso de sentencias **`if-if-else`** (si se quiere asociar el **`else`** al primer **`if`**).

La sentencia **`if-else if-else if`** .. se debe codificar con los **`else if`** alineados a la izquierda, para que refleje un comportamiento selectivo más que anidado:

```
if (expresion)
  {
    ...
  }
else if (expr2)
  {
    ...
  }
else if (expr3)
  {
    ...
  }
else
  {
    ...
  }
```

8. Otros

Es preferible realizar la comparación con 0 que con 1, ya que la mayoría de las funciones garantizan el cero si es falso, y no cero si es verdadero. Por lo tanto, suponiendo que FALSO y VERDADERO son 0 y 1 respectivamente:

```
if (VERDADERO == func())
```

se debe codificar como

```
if (FALSO != func())
```

Las asignaciones embebidas deben evitarse ya que hacen el código más difícil de leer, como en

```
d = (a = b + c) + r;       /* Incorrecto */
```

En cambio, cuando se trate de funciones de la biblioteca estándar que devuelvan un resultado que se utilice en la condición de una expresión condicional, puede resultar útil la llamada en la propia expresión, como en:

```
while (EOF != (c = getchar()))
  {
    /* procesa el carácter */
  }
```

Utilizar con precaución los operadores `++` y `--`, ya que pueden dar lugar a efectos laterales, código poco legible, etc. (ver un ejemplo en un apartado específico más abajo).

9. Organización en ficheros

El primer carácter del nombre del fichero debe ser letra, y el resto letras y números. Procurar usar siempre minúsculas.

El nombre de los ficheros fuente en lenguaje C deben tener como sufijo **`.c`** o **`.h`** (en minúsculas). Los ficheros con las definiciones de las funciones deben ser **`.c`** (si no es así el compilador no los acepta).

Se deben evitar ficheros de más de 1000 líneas.

Los ficheros **`.c`** NO deben ser incluidos en otros ficheros.

10. Ficheros de cabeceras

Contienen las definiciones de constantes, tipos definidos por el usuario y los prototipos de las funciones utilizadas en nuestro programa.

Tienen la extensión **`.h`** . Los ficheros de cabecera del sistema (p.e. `stdio.h`, `string.h`, etc.) se encuentran en un directorio predeterminado que el compilador conoce, por ello para incluirlos en nuestro programa no es necesario especificar la ruta de los ficheros (por ejemplo `#include <stdio.h>`).

Cada fichero de definición de funciones debe llevar asociado un fichero de cabecera, con los prototipos de las funciones que vayan a ser utilizadas en otros ficheros. Ambos ficheros deben tener el mismo nombre, pero el de definición terminado en `.c` y el de prototipos terminado en `.h`.

Cuando se va a definir un fichero de cabecera, no debe usar:

- los nombres de ficheros de cabecera del sistema para los de la aplicación.
- caminos absolutos para la inclusión de los ficheros de cabecera definidos por el usuario. Usar caminos relativos al directorio de trabajo entrecomillados.

Los ficheros de cabecera deben ser incluidos en el fichero que define la función o variable. De esta forma se hace comprobación de tipos, ya que el compilador comprueba que la declaración coincide con la definición (cada fichero `.c` incluye al menos a su correspondiente `.h`).

No se debe incluir en ellos definiciones de variables.

No se deben anidar ficheros de cabecera. Para evitar doble inclusión se deben utilizar las órdenes de preprocesado que hacen que se defina una etiqueta en el momento de la primera inclusión del fichero de cabecera. Por ejemplo, si se tuviera un fichero de cabecera de nombre `cuenta_palabras.h` con el prototipo de la función `cuenta_palabras`, debería quedar así:

```
#ifndef CUENTA_PALABRAS_H
#define CUENTA_PALABRAS_H

void cuenta_palabras(int argc,char **argv,
                     int *palabras,int *total);

#endif
```

De esta manera, si se intentara incluir de nuevo el mismo fichero de cabecera, al comprobar que la etiqueta (`CUENTA_PALABRAS_H`) ya está definida, el preprocesador se saltaría todo lo que hay hasta el final del fichero (donde está `#endif`).

Se deben declarar todas las funciones que se utilicen. Para usar funciones que están definidas en otros ficheros, se deben incluir los ficheros de cabecera correspondientes. Si existen funciones locales al fichero (sólo se usan en ese fichero) sus prototipos no irán en el correspondiente `.h`, sino que irán identificados claramente en el propio fichero `.c` .

11. La sentencia exit

Esta sentencia no se ha explicado en las clases teóricas. Y a pesar de ello, en algún momento, es posible sentirse tentado de usarla, posiblemente sin conocer muy bien su funcionamiento.

Y aunque no se ha prohibido su uso expresamente, sí se ha realizado de forma indirecta. Recordemos una de las bases de la programación modular:

> *Toda función tendrá un **único** punto de retorno, que además deberá ser la última línea de la función. Además, deberá devolver el control a la función que la invocó, y no a otra o al sistema.*

¿En qué consiste la sentencia **`exit`**? Esta sentencia provoca la interrupción del programa que se está ejecutando en el punto donde se encuentre, y permite devolver un valor al entorno desde el que se ejecutó el programa.

Ya podemos ver que esta sentencia rompe el flujo normal del programa, y hace que nuestro programa no sea modular.

Conclusión: no se puede utilizar la sentencia **`exit`**.

12. Los operadores `++` y `--`.

Mire el siguiente código:

```
/*
** Operador de incremento
*/
#include <stdio.h>

int main()
{
   int i = 7;

   printf("%d \n", i++ * i++);
   printf("%d \n", i);

   return 0;
}
```

Hay que evitar este tipo de expresiones. Se asegura que ++ provoca el incremento de la variable después de su uso. Este "después" es ambiguo, y el estándar lo deja indefinido. Sí aclara que se incrementa antes de ejecutar la siguiente sentencia. Luego el compilador puede optar por multiplicar `7*7`, o `7*8`.

El conocer cómo "nuestro" compilador realiza esta operación no debe servirnos de referencia, ya que no conocemos cómo se realiza en otros compiladores.

13. Orden de Evaluación

En la siguiente expresión:

```
f() + g() * h()
```

se puede asegurar que la multiplicación se realiza antes que la suma, pero no en qué orden se llama a las funciones.

Pero, ¿qué ocurre con los operadores lógicos `&&` y `||`? Son una excepción, y se asegura la evaluación de izquierda a derecha. Esto permite sentencias como:

```
while (EOF != (c = getchar()) && '\n' != c)
```

Además, en la evaluación de una expresión con operadores lógicos, se deja de evaluar cuando se puede asegurar el resultado de la expresión. Por ejemplo, en la siguiente condición, no se llama a la función comprueba si `p` vale `NULL` (ya que si se cumple esa condición la condición completa es falsa independientemente del valor que devolviera la función `comprueba`):

```
if (NULL != p && comprueba(*p))
  {
    ...
  }
```

14. De nuevo los Prototipos

A pesar de todo lo comentado hasta ahora, puede que no se haya convencido de la necesidad de declarar una función antes de utilizarla, es decir, que aparezca su prototipo antes de la llamada.

Para convencerle de su necesidad, se plantea el siguiente ejemplo:

```
/*
** Demostración de error debido a falta de prototipo.
** En este caso, al no existir prototipo, el
**   compilador supone que la funcion multip devuelve
**   un valor del tipo int.
*/

#include <stdio.h>

int main()
{
  double prod;
  int res;

  prod=multip(7.3, 3.5);

  printf("\n Prod= %f \n", prod);

  return 0;
}
```

La función `multip` está definida en otro fichero de la siguiente forma:

```
double multip(double a, double b)
{
  return a*b;
}
```

Si los compilamos y ejecutamos:

```
 %> gcc proto.c proto_aux.c
 %> a.out
```

Se puede observar que el resultado de la operación no es el esperado. Pruebe a compilar como:

```
%> gcc -Wall proto.c proto_aux.c
```

Se observa que ahora el compilador sí informa de estos problemas.