Sistemas Electrónicos Digitales Avanzados - Universidad de Alcalá

Hoja de estilos de programación en C

Versión 1.0 (20-septiembre-2013)

1. Hoja de estilos

Esta hoja de estilos se ha realizado en base a los conocimientos de programación de los autores y como recopilación de varias recomendaciones [1][2]

1.1. Nomenclatura

Para comentar código en un documento, las referencias se han de hacer utilizando los siguientes formatos:

- Ficheros – "fichero"
- Variables – 'variable'
- Funciones – *función*
- Código – `en este formato (courrier new, tamaño 8)`

1.2. Convenio de nombres

1.2.1. Variables

Los nombres de las variables es preferible que tengan el mismo nombre que la magnitud física que representan, y en caso de que no sean magnitudes físicas, es preferible aunque no necesario que sean en ingles, ya que el propio código viene del ingles y es complicado leer múltiples idiomas en paralelo, y el código se ha de escribir con la máxima legibilidad para permitir la exportación de código y la comprensión de este por otras personas. Es decisión del programador en todo caso dar un nombre, pero en todo caso se ha de optar por la máxima legibilidad del código.

Para dar nombres a variables se intentará dar un nombre claramente descriptivo del contenido de la variable, evitando utilizar nombres genéricos como 'temp', 'numero', etc.

Utilizando nombres descriptivos se avanza bastante ya que se auto documenta el código porque se entiende por si mismo sin necesidad de comentarios.

Para componer los nombres de las variables se debe utilizar una mezcla de mayúsculas y minúsculas, escribiendo en minúsculas y utilizando mayúsculas para iniciar las palabras a partir de la segunda, p.ej. 'sumaFinal'. Los nombres de los arrais deben ir en plural.

Para las declaraciones es mejor utilizar una sola sentencia, excepto en casos triviales, como la declaración de índices para bucles. Si el nombre de la variable no es suficientemente explicativo, se ha de añadir un comentario indicando el significado y uso de la variable.

Las variables se han de declarar inmediatamente antes de utilizarlas y lo mas localmente posible a su uso, evitando el uso de variables globales lo máximo posible.

Es preferible crear un nuevo entorno cuando una variable se va a utilizar para una o pocas operaciones, como pueden ser variables temporales, p. ej.:

```
{
    int finalSum;
    double dataReaded[DATA_NUMBER];

    {   // Inicio del bloque local
        int i;
        int auxiliar;
        auxiliar = firstElemet * lastElement;
        for (i = 0; i < DATA_NUMBER; i++)
        {
          finalSum = auxiliar * dataReaded[i];
        }
    }   // Final del bloque local
}
```

En este ejemplo se ve como las variables ‘i’ y ‘auxiliar’ solo se utilizan dentro del bloque comentado como local.

1.2.2. Defines

La definición de variables se debe hacer en un ficherote cabecera, agrupándolos por usos y utilizando únicamente mayúsculas un su declaración. Para separar palabras se ha de utilizar el carácter ‘_’.

Es preferible utilizar en el código valores definidos en lugar de numéricos, ya que al estar agrupadas las definiciones es más fácil cambiar el tamaño de un array o los valores a utilizar.

```
day = (3 + numberOfDays) % 7; // NO! uses literal numbers

#define WEDNESDAY = 3;
#define DAYS_IN_WEEK = 7;
day = (WEDNESDAY + numberOfDays) % DAYS_IN_WEEK; // Yes, self-documenting
```

1.2.3. Funciones

La composición de los nombres de las funciones es el mismo que para las variables, escribiéndolos en minúsculas y utilizando mayúsculas para iniciar las palabras a partir de la segunda.

Los nombres deben dar una explicación del contenido de la función, de modo que no sea necesario un comentario para describir lo que hace esa función.

Inicie el nombre de las funciones con un verbo que describa la acción a realizar. Si este verbo no fuera suficientemente descriptivo, añada un nombre. Añada un adjetivo si fuera necesario para clarificar el nombre.

Utilice los pronombres ‘get’ y ‘set’ para las funciones que realicen estas acciones sobre una variable. Las funciones con ‘get’ devuelven el valor de una variable, registro, etc. Las funciones ‘set’ cambian el valor de una variable, registro, etc.

Si la función devuelve un valor booleano, utilice los prefijos ‘is’ o ‘has’ y evite el uso de ‘not’. Es preferible utilizar el operador ‘!’ al llamar a la función.

Estos convenios son prácticas comunes que ayudan a hacer el código más legible para otras personas distintas a las que lo han escrito.

A pesar de que las funciones y variables se escriben en minúsculas seguidas por mayúsculas para diferenciar palabras, las funciones se distinguen principalmente por llevar paréntesis a continuación.

Es preferible que los argumentos que se pasan a una función se agrupen en estructuras cuando son funciones complejas y que esa estructura sea común a las principales funciones del código. Para funciones sencillas esto no es necesario.

1.2.4. Tipos y estructuras

Las definiciones de tipos se ha de hacer igual que las variables, pero empezando por mayúsculas. Las estructuras se definen finalizadas con ‘_t’, aunque es muy útil definir un tipo nuevo en la declaración.

```
typedef struct node_t
{
    void * content;
    struct node_t * next;
}Node;
```

1.3. Comentarios

Los comentarios añaden legibilidad al código describiendo partes de código, contenido de ficheros o descripción de funciones.

Al insertar un comentario, se ha de dejar un espacio entre el delimitador y el texto para facilitar la lectura de este.

Las operaciones complejas que no sean legibles han de ser comentadas explicando su funcionamiento. Hay que evitar comentar operaciones sencillas para reducir el tamaño de los ficheros.

Se deben utilizar cuatro tipos de comentarios:

- Comentarios de cabecera de fichero.
- Comentarios de partes de código.
- Descripciones de operaciones.
- Descripción de función.

Además se pueden utilizar temporalmente los comentarios '/*……..*/' para comentar partes de código en depuración que se quieran quita de la compilación. Para una versión final estas partes de código han de ser arregladas o eliminadas.

1.3.1. Comentarios de cabecera de fichero

Al inicio de los ficheros se ha de añadir un comentario en el que se muestre información acerca del contenido del fichero, así como la versión, la fecha y el autor.

También se debe añadir una sección de cambios entre versiones del fichero para identificar posibles fallos.

1.3.2. Comentarios de partes de código

Utilice estos comentarios para crear un sumario de las partes de código.

Evite comentar cada línea por separado, agrupándolas para realizar un solo comentario del contenido de las siguientes líneas.

Preceda cada línea de comentarios por una línea en blanco y alinee el comentario con el resto del código.

1.3.3. Descripción de operaciones

Cuando una operación o línea de código sea compleja de entender, se ha de añadir un comentario al final de la línea explicando esta operación.

Evite al máximo estos comentarios escribiendo código que se explique a si mismo. Es preferible reescribir el código de una forma mas clara, siempre y cuando no afecte a la ejecución de código en velocidad.

1.3.4. Descripción de función

Las funciones han de llevar una cabecera en la que se explique su utilidad, así como lo parámetros de entrada y valor de salida.

Estos comentarios se deben hacer para documentar el código, por lo que es interesante utilizar programas como Doxigen [3] para generar la documentación de forma automática del código programado. Este programa genera la documentación en base a una serie de etiquetas añadidas a los comentarios de las funciones.

1.4. Formato

Aplicar un formato definido aumenta la legibilidad del código. No se trata de ver si es mejor poner dos o tres espacios en ciertos sitios, sino de aplicar unas reglas y usarlas siempre. Aquí se describe una recomendación compuesta por varias reglas. Utilizando esto se puede aumentar la productividad y favorecer el cambio de programador sobre una misma aplicación.

1.4.1. Llaves y bloques

Para identificar los bloques se han de utilizar dos espacios mejor que tabulaciones, ya que las tabulaciones dependen del editor, y un mismo código leído con distintos editores puede cambiar su aspecto. Además, normalmente el tabulador ocupa mas espacio, lo que puede hacer con más facilidad que quede poco espacio dentro del marco del editor generando columnas de código muy estrechas.

Hay editores como puede ser emacs [1] que realizan este espaciado de forma automática facilitando esa labor al programador.

Hay que ser constante en la posición de las llaves de apertura y cierre de bloques. Esta posición corresponde a los límites del bloque.

```
void function (void)
{
  declaracion 1;
  declaracion 2;
  printf("hola mundo");
  for(i=0;i<7;i++)
  {
    printf("numero %d",i);
  }
}
```

Si un bloque se compone de una sola línea es permisible omitir las llaves, aunque si ese bloque contiene otro bloque de una sola línea y así sucesivamente es mejor poner las llaves para aumentar la legibilidad.

```
for(w = 0; w < with; w++)
  for(h = 0; h < height; h++)
    for(d = 0; d < depth; d++)
```

```
      a[w][h][d] = 0;

for(w = 0; w < with; w++)
{
  for(h = 0; h < height; h++)
  {
    for(d = 0; d < depth; d++)
    {
      a[w][h][d] = 0;
      b[w][h][d] = 1;
      c[w][h][d] = 2;
    }
  } // unnecessary, but more readable
} // unnecessary, but more readable
```

Se ha de evitar utilizar más de tres niveles en los bloques.

1.4.2. Espacios en blanco

Se ha de utilizar espacios en blanco y líneas en blanco para incrementar la legibilidad.

Utilice líneas en blanco para separar bloques de instrucciones que realizan una función específica. Estos bloques de código suelen ir precedidos de un comentario y tienen una longitud de entre 3 y 7 líneas.

Entre dos funciones se han de poner dos líneas en blanco para separarlas.

Se ha de usar un espacio en blanco:

- A ambos lados de los operadores matemáticos y símbolos de relación: c = a + b;
- Después de las comas.
- Después de los punto y coma en bucles for

No se ha de usar un espacio en blanco:

- Delante ni detrás de los operadores de selección (‘->’, ‘.’ , ‘[]’, etc)
- Delante ni detrás de los paréntesis.
- Después de un operador unitario (++, --, !, etc)
- Antes de un signo de puntuación.

1.4.3. Longitud de las líneas

Se ha de evitar utilizar líneas de más de 80 caracteres. Esta medida es la que tienen la mayoría de los editores de código. Cuando una línea supere esta longitud se ha de partir siguiendo las siguientes recomendaciones:

- después de una coma
- antes de un operador
- alinear la nueva línea con el principio de la operación en el mismo nivel que la línea inicial.

1.5. Estructuración del código. Modularidad

Se ha de restringir el acceso a las variables al mínimo. Una variable debe ser accesible solo en el bloque que se va a utilizar. Si una variable se va a utilizar solo en una línea, es mejor abrir un nuevo bloque en el que se declare la variable y se ejecute la instrucción que declararla al principio de la función.

Se ha de evitar la declaración de variables de forma global al máximo.

Se ha de organizar el código en unidades de compilación razonablemente pequeñas.

2. Bibliografía

[1] "A Guide to Coding Style", Justus Piater, 14-01-2005, (http://www.montefiore.ulg.ac.be/~piater/Cours/Coding-Style/index.html)

[2] "C♯ Coding Style Guide, Version 0.3", Mike Krüger, (http://sharpdevelop.net/TechNotes/SharpDevelopCodingStyle03.pdf)

[3] http://www.doxigen.org

[4] http://www.gnu.org/software/emacs/

3. Agradecimientos

Se agradece a D. Carlos Girón Casares la elaboración de la versión inicial de este documento en base a su experiencia como programador.