Fundamentos de Informática

Normas de Estilo para Programación en Lenguaje C

Independientemente de los algoritmos usados, hay muchas formas y estilos de programar. La legibilidad de un programa es demasiado importante como para prestarle la atención que merece por parte del programador.

Los programas, a lo largo de su vida, se van quedando obsoletos debido a cambios en su entorno. Un programa pensado para una determinada actividad, es muy normal tener que modificarlo porque cambie dicha actividad o porque decidamos incluirle nuevas posibilidades que antes no estaban previstas. De aquí las múltiples versiones que sacan al mercado las empresas de programación. Además, debido a la dificultad de algunos programas para probarlos exhaustivamente, a veces, se descubren errores cuando el programa lleva funcionando cierto tiempo.

Por otra parte, también es frecuente que uno deba modificar un programa escrito por otro programador. Es entonces, cuando hay que modificar un programa escrito hace cierto tiempo o escrito por otro programador, cuando salen los problemas de legibilidad de un programa. Para poder modificarlo, primero hay que comprender su funcionamiento, y para facilitar esta tarea el programa debe estar escrito siguiendo unas normas básicas. La tarea de **mantenimiento** del software (corregir y ampliar los programas) es una de las tareas más árduas del ciclo de vida del software. Por eso, al programar debemos intentar que nuestros programas sean lo más expresivos posibles, para ahorrarnos tiempo, dinero y quebraderos de cabeza a la hora de modificarlos.

Además, el **C** es un lenguaje que se presta a hacer programas complejos y difíciles de comprender. En C se pueden encapsular órdenes y operadores, de tal forma que, aunque consigamos mayor eficiencia su comprensión sea todo un reto. Como curiosidad, diremos que en EEUU existe un concurso de programación en C, en el que gana el que consiga hacer el programa más críptico.

Resumiendo y repitiendo, diremos que la claridad en un programa es de vital importancia, pues la mayor parte del tiempo de mantenimiento de un programa se emplea en estudiar y comprender el código fuente existente. Esto es especialmente importante cuando se trabaja en grupo pero no es exclusivo de este modo de trabajo. Naturalmente, esto es más importante cuantas más líneas de código tenga el programa, pero incluso en programas pequeños puede perderse mucho tiempo intentando comprender su funcionamiento, si no están programados con *mimo*.

Unas **normas de estilo** en programación, son tan importantes que todas las empresas dedicadas a programación imponen a sus empleados una mínima uniformidad, para facilitar el intercambio de programas y la modificación por cualquier empleado, sea o no el programador inicial. Por supuesto, cada programa

debe ir acompañado de una documentación adicional, que aclare detalladamente cada módulo del programa, objetivos, algoritmos usados, ficheros...

No existen un conjunto de reglas fijas para programar con legibilidad, ya que cada programador tiene su modo y sus manías y le gusta escribir de una forma determinada. Lo que sí existen son un conjunto de reglas generales, que aplicándolas, en mayor o menor medida, se consiguen programas bastante legibles. Aquí intentaremos resumir estas reglas.

1. Identificadores significativos

Un identificador es un nombre asociado a un objeto de programa, que puede ser una variable, función, constante, tipo de datos... El nombre de cada identificador debe *identificar* lo más claramente posible al objeto que identifica (valga la redundancia). Normalmente los identificadores deben empezar por una letra, no pueden contener espacios (ni símbolos raros) y suelen tener una longitud máxima que puede variar, pero que no debería superar los 10-20 caracteres para evitar lecturas muy pesadas.

Un identificador debe indicar lo más breve y claramente posible el objeto al que referencia. Por ejemplo, si una variable contiene la nota de un alumno de informática, la variable se puede llamar `nota_informatica`. Observe que no ponemos los acentos, los cuales pueden dar problemas de compatibilidad en algunos sistemas. El carácter `'_'` es muy usado para separar palabras en los identificadores.

Es muy normal usar variables como `i`, `j` o `k` para nombres de índices de bucles (`for`, `while`...), lo cual es aceptable siempre que la variable sirva sólo para el bucle y no tenga un significado especial. En determinados casos, dentro de una función o programa pequeño, se pueden usar este tipo de variables, si no crean problemas de comprensión, pero esto no es muy recomendable.

Para los identificadores de función se suelen usar las formas de los verbos en infinitivo, seguido de algún sustantivo, para indicar claramente lo que hace. Por ejemplo, una función podría llamarse `Escribir_Opciones`, y sería más comprensible que si le hubiéramos llamado `Escribir` o `EscrOpc`. Si la función devuelve un valor, su nombre debe hacer referencia a este valor, para que sea más expresivo usar la función en algunas expresiones, como:

```
Precio_Total = Precio_Total + IVA(Precio_Total,16) +
               Gastos_Transporte(Destino);
```

2. Constantes simbólicas

En un programa es muy normal usar constantes (numéricas, cadenas...). Si estas constantes las usamos directamente en el programa, el programa funcionará, pero es más recomendable usar constantes simbólicas, de forma que las definimos al principio del programa y luego las usamos cuando haga falta. Así, conseguimos principalmente dos ventajas:

- Los programas se hacen más **legibles**: Es más legible usar la constante

simbólica `PI` como el valor de p que usar 3.14 en su lugar:

```
Volumen_Esfera = 4/3. * PI * pow(radio,3);
```

- Los programas serán más **fáciles de modificar**: Si en un momento dado necesitamos usar `PI` con más decimales (3.141592) sólo tenemos que cambiar la definición, y no tenemos que cambiar todas las ocurrencias de 3.14 por 3.141592 que sería más costoso y podemos olvidarnos alguna.

En C, las constantes simbólicas se suelen poner usando una órden al Preprocesador de C, quedando definidas desde el lugar en que se definen hasta el final del fichero (o hasta que expresamente se indique). Su formato general es:

```
#define CONSTANTE valor
```

que se encarga de cambiar todas las ocurrencias de `CONSTANTE` por el valor indicado en la segunda palabra (`valor`). Este cambio lo realiza el preprocesador de C, antes de empezar la compilación. Por ejemplo:

```
#define PI 3.141592
```

Por convenio, las macros se suelen poner completamente en mayúsculas y las variables no, de forma que leyendo el programa podamos saber rápidamente qué es cada cosa. En general, se deben usar constantes simbólicas en constantes que aparezcan más de una vez en el programa referidas a un mismo ente que pueda variar ocasionalmente. Obsérvese, que aunque el valor de p es constante, podemos variar su precisión, por lo que es recomendable usar una constante simbólica en este caso, sobre todo si se va a usar en más de una ocasión en nuestro programa. Puede no resultar muy útil dedicar una constante para el número de meses del año, por ejemplo, ya que ese valor es absolutamente inalterable.

3. Comentarios, comentarios...

El uso de comentarios en un programa escrito en un lenguaje de alto nivel es una de las ventajas más importantes con respecto a los lenguajes máquina, además de otras más obvias. Los comentarios sirven para aumentar la claridad de un programa, ayudan para la documentación y bien utilizados nos pueden ahorrar mucho tiempo.

No se debe abusar de *comentarista*, ya que esto puede causar una larga y tediosa lectura del programa, pero en caso de duda es mejor poner comentarios de más. Por ejemplo, es absurdo poner:

```
Nota = 10; /* Asignamos 10 a la variable Nota */
```

Los comentarios deben ser breves y evitando divagaciones. Se deben poner comentarios cuando se crean necesarios, y sobre todo:

- Al **principio del programa** o de cada fichero del programa que permita seguir un poco la historia de cada programa, 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 que el programador quiera dejar constancia.
- En cada **sentencia o bloque** (bucle, `if`, `switch`...) que revista cierta complejidad, de forma que el comentario indique qué se realiza o cómo funciona.
- Al **principio de cada función** cuyo nombre no explique suficientemente su cometido. Se debe poner no sólo lo que hace sino la utilidad de cada parámetro, el valor que devuelve (si lo hubiera) y, si fuera oportuno, los requisitos necesarios para que dicha función opere correctamente.
- En la **declaración de variables y constantes** cuyo identificador no sea suficiente para comprender su utilidad.
- En los **cierres de bloques** con `'}'`, para indicar a qué sentencias de control de flujo pertenecen, principalmente cuando existe mucho anidamiento de sentencias y/o los bloques contienen muchas líneas de código.

No olvidemos que los comentarios son textos literarios, por lo que debemos cuidar el estilo, acentos y signos de puntuación.

4. Estructura del programa

Un programa debe ser claro, estar bien organizado y que sea fácil de leer y entender. Casi todos los lenguajes de programación son de formato libre, de manera que los espacios no importan, y podemos organizar el código del programa como más nos interese.

Para aumentar la claridad no se deben escribir líneas muy largas que se salgan de la pantalla y funciones con muchas líneas de código (especialmente la función principal). Una función demasiado grande demuestra, en general, una programación descuidada y un análisis del problema poco estudiado. Se deberá, en tal caso, dividir el bloque en varias llamadas a otras funciones más simples, para que su lectura sea más agradable. En general se debe modularizar siempre que se pueda, de forma que el programa principal llame a las funciones más generales, y estas vayan llamando a otras, hasta llegar a las funciones primitivas más simples. Esto sigue el principio de *divide y vencerás*, mediante el cual es más fácil solucionar un problema dividiéndolo en subproblemas (funciones) más simples.

A veces, es conveniente usar paréntesis en las expresiones, aunque no sean necesarios, para aumentar la claridad.

Cada bloque de especial importancia o significación, y cada función debe separarse de la siguiente con una línea en blanco. A veces, entre cada función se añaden una linea de asteriscos o guiones, como comentario, para destacar que empieza la implementación de otra función (aparte de los comentarios explicativos de dicha función).

El uso de la sentencia `GOTO` debe ser restringido al máximo, pues lía los programas y los hace ilegibles. Como dicen Kerninghan y Ritchie en su libro de C, su utilización sólo está justificada en casos muy especiales, como salir de una estructura profundamente anidada, aunque para ello podemos, a

veces, usar los comandos `break`, `continue`, `return` o la función `exit()`. Igualmente, esos recursos, para salir de bucles anidados deben usarse lo menos posible y sólo en casos que quede bien clara su finalidad. Si no queda clara su finalidad, será mejor que nos planteemos de nuevo el problema y estudiemos otra posible solución que seguro que la hay.

Normalmente, un programa en C se suele estructurar de la siguiente forma:

1. Primero los **comentarios** de presentación, como ya hemos indicado.
2. Después, la inclusión de **bibliotecas del sistema**, los ficheros `.h` con el `#include` y entre ángulos (`<...>`) el nombre del fichero. Quizás la más típica sea:

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

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

   ```
   #include "rata.h"  /* Rutinas para control del ratón        */
   #include "cola.h"  /* Primitivas para el manejo de una cola */
   ```

4. **Variables globales**, usadas en el módulo y declaradas en otro módulo distinto, con la palabra reservada `extern`.
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 de la función, no su implementación. De esta forma, el programa (y el programador) sabrá el número y el tipo de cada parámetro y cada función.
8. **Declaración de variables globales del módulo:** Se trata de las variables globales declaradas aquí, y que si se usan en otro módulo deberán llevar, en el otro módulo, la palabra `extern`.
9. **Implementación de funciones:** Aquí se programarán las acciones de cada función, incluida la función principal. Normalmente, si el módulo incluye la función principal, la función `main()`, ésta se pone la primera, aunque a veces se pone al final y nunca en medio. El resto de funciones, se suelen ordenar por orden de aparición en la función principal y poner juntas las funciones que son llamadas desde otras. Es una buena medida, que aparezcan en el mismo orden que sus prototipos, ya que así puede ser más fácil localizarlas.

Naturalmente, este orden no es estricto y pueden cambiarse algunos puntos por otros, pero debemos ser coherentes, y usar el mismo orden en todos los módulos. Por ejemplo, es frecuente saltarse la declaración de los prototipos de las funciones poniendo en su lugar la implementación y, por supuesto, dejando para el final la función `main()`.

Otro punto muy importante es el referente a variables globales. En general es mejor **no usar nunca variables globales**, salvo que sean variables que se usen en gran parte de las funciones (y módulos) y esté bien definida y controlada su utilidad. El uso de estas variables puede dar lugar a los

llamados *efectos laterales*, que provienen de la modificación indebida de una de estas variables en algún módulo desconocido. Lo mejor es no usar nunca variables globales y pasar su valor por parámetros a las funciones que estrictamente lo necesiten, viendo así las funciones como *cajas negras*, a las que se le pasan unos determinados datos y nos devuelve otros, perfectamente conocidos y expresados en sus parámetros.

Por la misma razón que no debemos usar variables globales, no se deben usar pasos de parámetros por referencia (o por variable), cuando no sea necesario.

5. Indentación o sangrado

La indentación o sangrado consiste en marginar hacia la derecha todas las sentencias de una misma función o bloque, de forma que se vea rápidamente cuales pertenecen al bloque y cuales no. Algunos estudios indican que el indentado debe hacerse con 2, 3 ó 4 espacios. Usar más espacios no aumenta la claridad y puede originar que las líneas se salgan de la pantalla, complicando su lectura.

La indentación es muy importante para que el lector/programador no pierda la estructura del programa debido a los posibles anidamientos.

Normalmente, la llave de comienzo de una estructura de control (`{`) se pone al final de la linea y la que lo cierra (`}`) justo debajo de donde comienza -como veremos más adelante- pero algunos programadores prefieren poner la llave `{` en la misma columna que la llave `}`, quedando una encima de otra. Eso suele hacerse así, en la implementación de funciones, donde la llave de apertura de la función se suele poner en la primera columna.

Sin embargo, poner la llave de apertura `{` al final de la línea que da comienzo a ese bloque es lo preferido por los programadores, pues uno puede "olvidarse" de ella sabiendo que el bloque empieza en esa línea y termina en el carácter `}` que haya justo debajo de ella. Además, poner el `{` al principio de la línea puede estorbar al hacer modificaciones.

Veamos, a continuación, la indentación típica en las estructuras de control:

Sentencia `if-else`

```
if (condición) {
   sentencia1;
   sentencia2;
   ...
}
else {
   sentencia1;
   sentencia2;
   ...
}
```

Sentencia `swith`

```
switch (expresión) {
  case expresión1: sentencia1;
```

```
                    sentencia2;
                    ...
                    break;
  case expresión2: sentencia1;
                    sentencia2;
                    ...
                    break;
  .
  .
  .
  case default   : sentencia1;
                    sentencia2;
                    ...
}
```

Sentencia `for`

```
for (exp1;exp2;exp3) {
   sentencia1;
   sentencia2;
   ...
}
```

Sentencia `while`

```
while (condición) {
   sentencia1;
   sentencia2;
   ...
}
```

Sentencia `do-while`

```
do {
   sentencia1;
   sentencia2;
   ...
} while (condición);
```

Aunque estos formatos no son en absoluto fijos, lo que es muy importante es que quede bien claro las sentencias que pertenecen a cada bloque, o lo que es lo mismo, donde empieza y termina cada bloque. En bloques con muchas lineas de código y/o con muchos anidamientos, se recomienda añadir un comentario al final de cada llave de cierre del bloque, indicando a qué sentencia cierra. Por ejemplo:

```
for (exp1;exp2;exp3) {
   sentencia1;
   sentencia2;
   ...

   while (condición_1) {
      sentencia1;
      sentencia2;
      ...
      if (condición) {
         sentencia1;
         sentencia2;
         ...
```

```
         while (condición_2) {
           sentencia1;
           sentencia2;
           ...
         } /*while (condición_2)*/

      } /*if*/
      else {
         sentencia1;
         sentencia2;
         ...

         if (condición) {
            sentencia1;
            sentencia2;
            ...
         }
         else {
            sentencia1;
            sentencia2;
            ...
         }

      } /*else*/
   } /*while (condición_1)*/
} /*for*/
```

6. Presentación

Al hacer un programa debemos tener en cuenta quien o quienes van a usarlo o pueden llegar a usarlo, de forma que el intercambio de información entre dichos usuarios y el programa sea de la forma más cómoda, clara y eficaz posible.

En general, se debe suponer que el usuario no es un experto en la materia, por lo que se debe implementar un interfaz que sea fácil de usar y de aprender, intuitivo y que permita efectuar la ejecución de la forma más rápida posible.

Para ello, se suelen usar las siguientes técnicas:

- Usar **argumentos en la línea de comandos** para especificar las distintas opciones posibles.
- Avisar lo más detalladamente posible de todos los **errores** que se produzcan, los motivos de los mismos y cómo solucionarlos.
- Incluir una **ayuda** en línea (*on line*) en el programa, de forma que el usuario pueda consultar, en cualquier parte del programa, cómo ejecutar cada una de las acciones posibles en cada momento. Un estándar es usar la tecla de función F1 para solicitar esta ayuda. Es muy usual incluir el argumento ? (o con una barra o guion delante) para solicitar que el programa nos muestre una ayuda sobre su funcionamiento, opciones, ficheros usados... Si el programa requiere algunos argumentos obligatorios, se puede mostrar esa ayuda si falla alguno de esos argumentos.
- Usar **teclas estándar**: ESC para salir o deshacer las modificaciones

hechas, `RePág` y `AvPág` para retroceder y avanzar una página, `Supr` (o `Del`) para borrar, las flechas para seleccionar o cambiar de posición, el tabulador...

- Usar las **teclas de función** (`F1`, `F2`...) para las operaciones más usuales y combinaciones de teclas para acceder a otras operaciones de forma rápida (como `Alt-P`, `Ctrl-F1`...).
- Cualquier programa debe ir siempre acompañado de un **manual de usuario**, en el que se explique detalladamente todo lo referente a la ejecución del programa: Requisitos mínimos de ejecución (de procesador, memoria RAM necesaria, espacio en disco que ocupa...), parámetros disponibles, ficheros que usa, mensajes de error, soluciones a los errores, variables de entorno (si las usa), ejemplos (si procede)...
- La ejecución y manejo de programas se ha facilitado mucho con la llegada de **entornos gráficos**, en los que se usan comúnmente menús de opciones, iconos (pequeños dibujos), botones... Además, el manejo del **ratón** (*mouse*) hace más intuitivo y cómodo el manejo de determinados programas (aunque éstos no sean ejecutados bajo un entorno gráfico).
- Permitir al usuario **volver hacia atrás**: Es muy frecuente que, ejecutando un programa, se necesite modificar un dato que hemos indicado anteriormente o que simplemente deseemos ver algo anterior. Un programa debe permitir volver a cualquier punto anterior para mostrar o correquir cualquier dato. Para ello es muy frecuente usar la tecla `ESC` (carácter `'\x1B'` en un PC).
- **Indicar lo que se está haciendo:** Un programa no debe permitir pasar mucho tiempo efectuando alguna operación sin indicar al usuario qué está haciendo, cuanto le queda o la fase que está actualmente ejecutando. En programas de grandes cálculos es muy necesario para evitar que el usuario se desespere y piense que algo ha fallado. Se debe, por supuesto, incluir siempre la posibilidad de parar el proceso.
- **Avisar de los cambios importantes y confirmar la acción si es necesario:** Cuando el usuario elige alguna acción, si ésta es delicada o no admite vuelta atrás es necesario avisarle al usuario de lo que se va a hacer y exigirle que confirme tal acción. Por ejemplo, al borrar un fichero o alterar datos importantes.