Lenguaje de programación C

versión 1.2 10-7-2018

1. Entre este guión y el resto de ficheros del directorio se pueden ir realizando los ejercicios que se proponen más abajo. El punto 2 puede ser un poco difícil de entender por completo al principio y no es imprescindible para realizar el resto de puntos, pero es importante tenerlo claro para comprender mejor el proceso de realización de un programa.
2. El fichero `nombre_fichero.c` contiene el código fuente en el lenguaje C en formato ASCII entendible por personas. Al compilarlo lo traducimos a código máquina entendible solo por ese tipo ordenador (el tipo de ordenador depende del procesador y del sistema operativo). El fichero que obtenemos, `nombre_fichero.o`, se llama código objeto. Al enlazarlo obtenemos el fichero `nombre_fichero` que es el ejecutable, el programa que va a realizar lo que hemos programado. Al enlazar unimos a nuestro código objeto otros códigos objetos necesarios para que funcione nuestro programa, como por ejemplo la función `printf` que imprime por pantalla. Por tanto el código objeto y el código ejecutable son código máquina. Compilar y enlazar también se puede hacer en un solo paso en cuyo caso no guardamos el código objeto. Los errores de compilación son diferentes de los de enlazado. Un error de compilación puede ser un error de sintaxis del lenguaje y un error de enlazado puede ser que falte el código objeto de una función. El programa que compila y enlaza se llama compilador, nosotros usaremos el `gcc` de GNU.
3. Analiza, compila y ejecuta `hola.c`. Para analizarlo usa el editor de texto `gedit` que se encuentra en `Aplicaciones > Accesorios`. Si no te muestra en colores las diferentes partes del código configúralo en `Ver > Modo resaltado`. Para compilar y ejecutar usa la `terminal`, que se encuentra también en `Aplicaciones > Accesorios`.
4. Analiza, compila y ejecuta `entrada_salida_numero.c`.
5. Analiza, compila y ejecuta `doble.c`.
6. Realiza un programa que pida un número y que le sume 3. Llámalo `suma_3.c`.
7. Realiza un programa que pida un número y que le reste 3. Llámalo `resta_3.c` Usa `gedit` para editar el código y pulsa `Ctrl+s` para guardar y `Alt+Tabulador` para cambiar entre la ventana de `gedit` y la de `terminal`. Puedes tener varias pestañas en `gedit` para reutilizar el código de otros programas y con `Alt+1, 2, 3` cambias entre la primera pestaña, segunda, tercera, etc.
8. Realiza un programa que pida un número y que lo multiplique por 3. Llámalo `multiplica_3.c`.
9. Realiza un programa que pida un número y que lo divida entre 3. Llámalo `divide_3.c`.

10. Realiza un programa que pida un número y que lo eleve al cubo. Llámalo `exponencia_3.c`. Para elevar `x` a `y` hay que usar la función `pow (x, y)`, hay que incluir el archivo `math.h` y hay que enlazar con las librería matemáticas con la opción `-lm`, para ello:

    ```
    compilar: gcc -c nombre_fichero.c -o nombre_fichero.o
    enlazar: gcc nombre_fichero.o -o  nombre_fichero -lm
    ```

    o todo en uno: `gcc nombre_fichero.c -o  nombre_fichero -lm`

11. Realiza un programa que pida dos números reales y que los sume, reste, multiplique, divida y exponencie. Llámalo `operaciones.c`.

12. Expresión: conjunto de datos y operadores. Al evaluarlos su resultado es un número. Por ejemplo:

    `2 * x + 3` usa los operadores aritméticos `*` y `+`

    `x > 0` usa el operador de comparación `>`

    `x = x + 2` usa el operador de asignación `=` y el operador aritmético `+`

13. Sentencia: puede ser de expresión o de palabra clave.

    Sentencia de expresión: consta de una expresión acabada con punto y coma. Por ejemplo:

    ```
    2 * x + 3;
    x > 0;
    x = x + 2;
    ```

    Sentencia de palabra clave: consta de una palabra clave del lenguaje y dependiendo de cuál sea la palabra clave finaliza con punto y coma o no. Por ejemplo:

    ```
    float x;
    printf("Hola");
    return;
    if (x > 0) {
      z = z + 2;
      i = 3;
    }
    ```

    como vemos la sentencia condicional `if` no finaliza con punto y coma pero incluye sentencias de expresión.

14. Analiza, compila y ejecuta el programa `mayor_que_0.c` que solicita un número entero y dice si es mayor que 0 o si no lo es. Este programa usa la sentencia de control:

    ```
    if ( expresión ) {
    ```

```
   sentencia1
   sentencia2
}
else {
   sentencia3
   sentencia4
}
```

Si solo hay una sentencia se pueden obviar las llaves:

```
if ( expresión)
   sentencia1
else
   sentencia2
```

La `expresión` es la condición. Al evaluar dicha `expresión`, 0 es falso y cualquier otro valor es cierto. Los operadores disponibles son: `>` mayor que, `<` menor que, `>=` mayor o igual que, `<=` menor o igual que, `==` igual que, `!=` distinto que.

15. Realiza un programa llamado `mayor_igual_que_0.c` que solicite un número entero y diga si es mayor o igual que 0.
16. Realiza un programa llamado `mayor_igual_que_3coma5.c` que solicite un número cualquiera (real) y diga si es mayor o igual que 3,5. Usa el tipo `float`.
17. Realiza un programa llamado `mayor_menor_igual_que_0.c` que solicite un número entero y diga si es mayor que 0, menor que 0 o igual que 0.
18. Realiza un programa llamado `compara.c` que solicite dos números y diga cuál es el mayor.
19. Realiza un programa llamado `par_impar.c` que solicite un número y diga si es par o impar. La función `%` calcula el resto entre dos números, ejemplo `9 % 2` dará un valor `1`, y `8 % 2` dará un valor `0`.
20. Analiza, compila y ejecuta el programa `suma_1a100.c` que calcula el resultado de sumar todos los números del 1 al 100 (ambos inclusive). Este programa usa la sentencia de repetición bucle `for`:

    ```
    for (expresión1; expresión2; expresión3) {
      sentencia1
      sentencia2
    }
    ```

    la `expresión 2` es la condición, si es cierta (diferente de cero) se hace una iteración.

    Habitualmente usamos el bucle `for` con una variable como variable contador, en este ejemplo la variable `i` es la variable contador:

```
for (i = 1; i <= 100; i++) {
  sentencia1
  sentencia2
}
```

Y si solo hay una sentencia podemos no poner llaves, de esta manera:

```
for (i = 1; i <= 100; i++)
  sentencia1
```

o de esta otra:

```
for (i = 1; i <= 100; i++) sentencia1
```

la primera suele ser más clara.

21. Realiza un programa que multiplique todos los números del 1 al 100 (ambos inclusive). Llámalo `multiplica_1a100.c`.
22. Realiza el programa `suma.c` que pida dos números y luego calcule la suma de todos los números comprendidos entre los solicitados, incluidos ellos mismos. Incluye código para que funcione en los siguientes casos: los dos números son iguales, el usuario puede introducir los números en cualquier orden, primero el menor o primero el mayor.
23. Realiza un programa llamado `factorial_10.c` que calcule el factorial del número 10. Puedes compilar con la opción `-Wall`, que te muestra todos los avisos (warnings). Los avisos no son errores de compilación (por ejemplo, te avisa si has declarado una variable que luego no usas) pero te pueden ayudar a encontrar los errores lógicos, que son los más difíciles de encontrar.
24. Realiza un programa llamado `factorial.c` que calcule el factorial de un número. Introduce código de error para cuando el usuario introduzca números negativos. Cuando un programa no funciona como nosotros pretendemos, el método más sencillo de depuración es poner `printf` en sitios estratégicos para ver el valor de las variables. También podemos usar un depurador como el programa `gdb` (pequeño manual de uso en:

    http://www.lsi.us.es/~javierj/ssoo_ficheros/GuiaGDB.htm). No obstante no debemos pensar que un depurador va a resolver un programa mal planteado. Es más eficaz analizar detenidamente el programa que usar un depurador mediante ensayo y error.
25. Realiza un programa que indique si un número es primo. Llámalo `primo.c`. Usa un bucle `while` (busca información de su sintaxis).
26. Realiza un programa que calcule los n-primeros números primos. Llámalo `n_primos.c`.
27. Realiza un programa que calcule los números primos hasta el número `n` (incluido). Llámalo `primos_hasta_n.c`.

28. El estilo de indentar (sangrado) las partes del programa es libre pero se suelen seguir varios estilos: el de GNU, el del kernel de Linux, el K&R (Kerninghan y Ritchie), y el de Don E. Knuth. Los tres últimos se diferencian solo en el número de espacios, ocho, cuatro y dos respectivamente. Por ejemplo el de K&R quedaría así:

```
if (i == 0) {
    x = 2;
    y = 1.5;
} else {
    x = 3;
    y = 0;
}
```

El mismo código con el estilo GNU (este estilo es muy claro pero a costa de gastar muchas líneas) es:

```
if (i == 0)
  {
    x = 2;
    y = 1.5;
  }
else
  {
    x = 3;
    y = 0;
  }
```

Se puede cambiar fácilmente de un estilo a otro usando el programa indent:

`indent nombrefichero.c` lo cambia a GNU

`indent -kr nombrefichero.c` lo cambia a K&R

`indent -linux nombrefichero.c` lo cambia a Linux

Además `indent` deja el código más legible, si por ejemplo escribimos: `x=y+z`, `indent` nos los escribe así: `x = y + z` (Evidentemente también lo podemos escribir nosotros directamente y no tener que usar `indent`).

También es conveniente dejar las líneas a menos de 80 columnas. De esta manera, al imprimir con un editor de texto plano, las líneas no salen cortadas (algunas impresoras imprimirán hasta la columna 93, otras hasta la 103...). También garantizamos que el código se pueda leer fácilmente en cualquier pantalla (incluso en terminales de 80 columnas) y aumentamos la legibilidad del código al no tener que leer líneas muy largas (el comando `fold` nos corta las líneas a 80 columnas: `fold -s nombre_fichero.txt`). También permitimos el

envío de código por correo electrónico puesto que el estándar de correo no permite líneas más largas de 80 columnas por lo que si enviamos código con líneas largas los servidores de correo pueden recortar dichas líneas y obtenerse resultados erróneos al compilar el programa.