PRO – 13/12/2019
EXAMEN UD03 (I)

NOMBRE: **David Bernal Navarrete**

---

1. Indica el grado de veracidad de la siguiente frase explicándola en su totalidad y razonando detalladamente sobre éste: "*Una alta cohesión facilita el mantenimiento*".

   Empezando con conceptos: la cohesión es el grado de interconexión que tienen los elementos de un módulo o función; el mantenimiento es la corrección de errores y cambios que se deban hacer al programa para garantizar su correcto funcionamiento. Cuando se habla de una "alta cohesión", se implica que los elementos de dicha función están fuertemente interconectados, lo que indica que resuelven un problema único. Debido a esta alta dependencia entre los elementos de una función, si se requiriera cambiar o corregir una parte del programa, sería un trabajo más sencillo, ya que los cambios pertenecen a una sola función, con un funcionamiento sencillo.
   Aunque una alta cohesión facilita el mantenimiento, solo lo facilita si se tiene en cuenta el acoplamiento entre las diferentes funciones. El acoplamiento es el grado de interconexión entre las diferentes funciones. Para que un programa esté bien diseñado, se debe aspirar a una alta cohesión, y a un mínimo acoplamiento. Al tener funciones únicas, cada una solucionando un problema concreto, se consigue una mayor eficiencia. Además, encontrar errores y corregirlos, o hacer cambios, se hace mucho más fácil, ya que si un error pertenece a una función y no afecta a las demás, la corrección se realiza rápida y eficazmente. Esta combinación de alta cohesión y mínimo acoplamiento también permite que el código sea más fácil de entender y de depurar.
   En conclusión, la frase está incompleta, ya que le falta indicar que es el conjunto de alta cohesión y mínimo acoplamiento el que facilita el mantenimiento en gran medida.

2. Prototipado modular. Concepto, necesidad, elementos (obligatorios y opcionales) que lo componen.

   El prototipado modular consiste en indicar al programa que existe una función antes de que esta sea definida. Si no se usa el prototipado, habrá que definir las funciones antes de que se les invoque en el programa principal. Aunque parece tan simple como definirlas antes del programa principal, en las ocasiones en las que una función haga uso de otra, deberán ir en un orden concreto, y en programas complejos con multitud de funciones esto puede ser una tarea difícil de realizar. El prototipado permite instanciar las funciones antes de definirlas, para indicarle al programa que si se encuentra una llamada a una función que todavía no ha sido definida, no se tratará de un error, sino que se definirá en otro punto del programa. Esto permite al programador despreocuparse del orden de definición de las funciones.
   El prototipado tiene la siguiente sintaxis:

   ```
   tipo nombre_funcion(tipo parametro_1, tipo parametro_2, ..., tipo parametro_n);
   ```

   Los elementos obligatorios son: el tipo de la función, el nombre de la función, el tipo del parámetro y el orden en el que deben ir. Por supuesto, olvidar el punto y coma del final de la sentencia resultará en horas perdidas buscando el ";" que falta. Por último, el nombre del parámetro de entrada es opcional.
   Un ejemplo de prototipado sería el siguiente:

   ```
   int suma(int a, int b);
   ```

   Siendo opcionales los nombres de parámetros "a" y "b".

3. Sobrecarga del operador * (asterisco). Explicación y ejemplos.

El operador * puede hacer referencia a tres cosas en el lenguaje C: puede ser el operador de multiplicación; puede indicar que se acceda al valor almacenado en una dirección de memoria; o indica la definición de un puntero. Un ejemplo para aclarar las tres posibilidades es el siguiente:

```
int *p;    //Un puntero "p" que indica a un entero
a = 2*4;  //Esto es una multiplicación
a = *b * 2; //Esto indica 1) que b es un puntero que indica a una dirección de memoria y 2) que va a
            // acceder al valor almacenado en esa dirección y lo va a multiplicar por 2.
```

Se podría declarar un puntero que indica a la dirección de memoria de un valor:

```
int b = 2;
int *p_b = &b;
```

El puntero "p_b" almacena la dirección de memoria del entero "b".
El operador * también se puede usar en la declaración de una función, tal que así:

```
void asignar_numero(int a, int *b);
```

Esto indica que en el segundo parámetro espera una dirección de memoria.

4. Contesta razonadamente estas preguntas sobre la instrucción `return`:

   a) ¿Puede una función sin parámetro de salida tener un `return` en su código?
   No, esto daría error. El compilador no espera ninguna salida de esta función, así que una instrucción que indica una salida en dicha función será inesperada y dará error.
   b) ¿Puede una función con parámetro de salida tener más de una instrucción `return` en su código?
   Sí, si las diferentes instrucciones 'return' se encuentran en diferentes brazos de una estructura selectiva. Sin embargo, varias instrucciones 'return' en una misma sección del código resultará en el compilador registrando más de una salida a una función que solo puede tener una, lo cual resulta en un error de compilación.
   c) ¿Puede una función con parámetro de salida no tener ninguna instrucción `return`?
   No, dará error. Ocurre lo mismo que en el apartado a): el compilador espera un valor de salida, y al no recibir ninguna instrucción 'return', el compilador estima que falta dicha instrucción, y resulta en un error de compilación.

Indicar en todos los casos cuál es el comportamiento del compilador.

5. Dado el siguiente esquema de programa:

```
void leer_numero (int *n)
{
  scanf("%d", n);
}

void pedir_numero (int *valor, int *positivo)
{
  printf("Introduzca un numero:");
  leer_numero (valor);
  if (*valor < 0) *positivo = 0;
}

void main ()
{
  int num, signo = 1;
  int *n;
  pedir_numero (&num, &signo);
  n = *num % 10;
  printf("La última cifra es %d", n);

  /*
    . . . . . .
    Resto de instrucciones del programa
    . . . . . .
  */
```

```
}
```

Realizar las siguientes tareas:

a) Indicar si hay algún error semántico en el uso de las variables y parámetros y, en su caso, proponer las modificaciones oportunas.

En la función leer_numero() no se realiza una limpieza del búfer de entrada tras el scanf.
En la función main(), en la cuarta instrucción se está intentando almacenar en un puntero un valor. A un puntero solo se le puede asignar una dirección de memoria, y el valor que haya en esa dirección debe ser del mismo tipo que el puntero. Además de que se está intentando asignar un valor a un puntero, *num está intentando acceder al valor almacenado en la dirección de memoria 'num'. 'num' no es una dirección de memoria, es un valor, por lo que esto es otro error..
En la siguiente instrucción (el printf) se intenta formatear un decimal (%d) con 'n', que de nuevo, es un puntero, y almacena una dirección de memoria, no un entero decimal. Esta instrucción también daría error.

Las correcciones serían las siguientes:
En la función leer_numero(), añadir tras el scanf la instrucción "fflush(stdin); ".
En la función main(), debido a que el puntero 'n' no se usa para nada, declararlo como un entero, sustituyendo "int *n;" por "int n;".
Al hacer esto, en la instrucción "n = *num % 10;" solo quedará el error de *num, por lo que se elimina el operador * y quedaría tal que así: n = num % 10; , con lo que conseguiríamos almacenar en n la última cifra del numero introducido. Al haber cambiado el tipo de n, el formato en el printf con un entero decimal ya no dará error, y mostrará por pantalla la última cifra del entero num.
Una versión modificada con estos cambios se encuentra en el apartado c).

b) Teniendo en cuenta el apartado anterior, describir gráficamente el uso de memoria de las variables utilizadas por el programa y por las funciones que utiliza (con las correcciones propuestas).

Cada línea hará referencia a una dirección de memoria diferente, con su valor en medio y el nombre de la variable a la derecha.
Primero, las variables que se definen en main():

Dirección de mem		Valor	Nombre de variable
dir_num (&num)	---	valor_num	--- num
dir_signo(&signo)	---	1	--- signo
dir_n (&n)	---	valor_n	--- n

Los valores de num y de n dependerán del valor introducido en la función leer_numero().
En la función main(), al invocar a la función pedir_numero(), se le dan los dos parámetros "&num" y "&signo". Esto es debido a que la función pedir_numero() espera dos direcciones de memoria como parámetros, y además cambiará el valor almacenado en &signo dependiendo de si el valor introducido es positivo o negativo, lo cual lo comprueba usando *valor, que hace referencia al valor almacenado en la dirección de memoria de &num (valor el cual es num).
Sin embargo, todavía no conocemos el valor que hay en &num, por esto, en pedir_numero() se invoca a la función leer_numero(), la cual recibe la dirección de memoria de num, y escribe en ella el valor dado por teclado. Debido a que la función leer_numero() espera una dirección de memoria como parámetro, al hacer el scanf (que usa una dirección de memoria para almacenar el dato) no es necesario acompañar al parámetro n del ampersand (&).

Digamos que introducimos el valor 12. Entonces la tabla de direcciones anterior se vería así:

Dirección de mem		Valor	Nombre de variable
dir_num (&num)	---	12	--- num
dir_signo(&signo)	---	1	--- signo
dir_n(&n)	---	2	--- n

c) Describir el valor de las variables y parámetros en todas aquellas instrucciones en que son utilizados.

Enumeraré las instrucciones en el orden en el que son ejecutadas, y para cada número indicaré el valor de las variables y parámetros, en el caso de que los haya.

```
void leer_numero (int *n)
{
  scanf("%d", n);                    // 6
   fflush(stdin);                    // 7
}

void pedir_numero (int *valor, int *positivo)
{
  printf("Introduzca un numero:");   // 4
  leer_numero (valor);               // 5
  if (*valor < 0) *positivo = 0;     // 8
}

void main ()
{
  int num, signo = 1;                // 1
  int n;                             // 2
  pedir_numero (&num, &signo);               // 3
  n = num % 10;                              // 9
  printf("La última cifra es %d", n);        // 10
}
```

1. num no tiene valor, signo es igual a 1.
2. n no tiene valor.
3. &num y &signo son las direcciones de memoria de dichas variables, supongamos las direcciones 01 y 02 (respectivamente) como ejemplo.
4. No hay ninguna variable ni parámetro.
5. ‘valor’ es un puntero que indica a la dirección de memoria de num: 01.
6. n es la dirección de memoria que se ha dado como parámetro. El parámetro era el puntero ‘valor’, que indica a la dirección de memoria 01. El scanf sobrescribirá el valor almacenado en 01 por el introducido por teclado. Como ejemplo, se ha escrito el valor 12.
7. stdin indica el búfer de entrada.
8. *valor indica al valor almacenado en la dirección de memoria a la que apunta el puntero ‘valor’: la dirección de memoria a la que apunta ‘valor’ es 01, y el valor almacenado en ella es 12. Por lo que *valor < 0 evalúa a ‘falso’, por lo que no se cambia el valor almacenado en la dirección de memoria a la que indica el puntero ‘positivo’ (indica a la dirección 02), que es el valor de la variable ‘signo’ (1).
9. Se le asigna al entero ‘n’ el valor de la variable num (12) en módulo 10. Esto da como resultado 2.
10. n es igual a 2.

6. Escribir el código fuente de la siguiente función:

```
void maquina_del_tiempo (int *dia, int *mes, int *anio, int *hora, int *minutos,
                         int días_extra, int min_extras)
```

La función recibe (por referencia) una fecha y una hora, y (por valor) un número de días y de minutos que hay que avanzar el reloj-calendario. Lo que debe hacer es "devolver", a través de los parámetros, los valores de dicho reloj-calendario una vez hechos los cambios.

Se debe suponer que:

- Todos los datos han sido ya validados con anterioridad a la correspondiente llamada.
- Se considera que todos los meses son de 30 días.

Se pueden/deben utilizar las funciones auxiliares que se estimen necesarias.

```
void maquina_del_tiempo (int *dia, int *mes, int *anio, int *hora, int *minutos, int dias_extra, int
                                    min_extras)
{
    // Primero vamos a extraer los años y meses que adelantaremos de dias_extra.
    int anio_extra = dias_extra % 365;
    dias_extra = dias_extra / 365;
    int mes_extra = dias_extra % 30;
    dias_extra = dias_extra / 30

    // Segundo, vamos a extraer las horas de min_extra
    int horas_extra = min_extras % 60;
    min_extras = min_extras / 60;

    // Tercero, vamos a cambiar los valores del calendario (*dia, *mes, y *anio)
    // Usaremos *(variable) para acceder al valor almacenado en esa dirección de memoria
    *anio = *anio + anio_extra;
    *mes = *mes + mes_extra;
    *dia = *dia + dias_extra;

    // Cuarto, haremos lo mismo pero para los valores del reloj (*minutos y * hora)
    *hora = *hora + horas_extra;
    *minutos = *minutos + min_extras,

    /*
        Tras la ejecución de las anteriores instrucciones, los valores almacenados en las direcciones de
        memoria dadas como parámetros a la función habrán sido cambiados, lo que "devuelve" los
        valores del calendario y del reloj cambiados.
    */
}
```