



Apellido y Nombres	Legajo	# de Hojas	Profesor

Normas Generales

Numere las hojas entregadas. **Lea detenidamente cada punto; la interpretación del enunciado forma parte de la evaluación.**

Complete en la primera hoja la cantidad total de hojas entregadas. Realice este parcial con tinta color azul o negro. **No utilice rojo ni verde por favor.** Cada ejercicio debe realizarse en hojas separadas y numeradas. Debe identificarse cada hoja con: Nombre, Apellido, Legajo. **Por favor entregar esta hoja y las restantes del tema junto al examen.**

Sección teórica – 20 minutos

1. Considere el siguiente programa:

```
1 #include <stdio.h>
2 #define TAM 7
3
4 int main(void)
5 {
6     short v[TAM]={72, 0x6F, 120, 97, 0x80, 2, 50};
7     short *p, *q;
8     p=v;
9     q=&v[TAM-2];
10
11    *(p+1)=104;
12    *p=v[0]^11;
13    *(v+2)=p[3];
14    3[v]=*(p+2)|0x55;
15    *q=p[4];
16    v[*(p+4)-45]=51;
17
18    printf("\n\n");
19    return 0;
20 }
```

Indique cómo evoluciona el contenido del vector **v** desde su inicialización en la línea 6 a medida que se ejecutan las líneas 11 a 16 (ambas inclusive).

Si considera que alguna línea tiene un error de cualquier tipo, explique el error.

Sección Práctica – 1 hora 15 minutos

1. Se tiene de un archivo binario con varios registros, que responden a la siguiente estructura:

```
struct datos
{
    int user_code;
    char user_name[15];
    float user_fee;
};
```



Apellido y Nombres	Legajo	# de Hojas	Profesor

Lamentablemente, al intentar accederlo, se comprobó que algunos de los registros estaban corrompidos.

Sin embargo, se dispone de un archivo verificador, en el que se almacenó un byte verificador para cada uno de los registros del archivo. Hay varias formas de obtener ese byte verificador; para este caso se generó haciendo un “acumulador de exor”; es decir, se hizo un exor entre los dos primeros bytes de un registro, a ese resultado parcial se le hizo exor con el tercer byte, luego con el cuarto, etc. Esto se denomina “paridad”

Como ejemplo, considere este registro de 5 bytes:

registro	[0]	[1]	[2]	[3]	[4]
----------	-----	-----	-----	-----	-----

El byte verificador (o de paridad) se obtendría de la siguiente manera:

Byte_paridad = registro[0] ^ registro[1] ^ registro[2] ^ registro[3] ^ registro[4];

Ahora bien, ¿cómo detecto los errores? Obteniendo el byte verificador del registro a verificar y comparándolo con el que se encuentra en el archivo verificador. Si coinciden, el registro está OK. Si no coinciden, es prueba de que el registro se corrompió.

En base a lo expuesto, le pedimos lo siguiente:

- a) Escriba la función

unsigned_char paridad(char *s, int c);

Propósito: hallar el byte de paridad de un string
Recibe: **s**: string al que debe hallarse la paridad
c: tamaño del string

Devuelve: el byte de paridad (acumulado de exor de todos los bytes)

- b) Escriba la función (use solo call systems (funciones de archivo de bajo nivel))

int verif(char *f1, char *f2);

Propósito: compara el byte de verificación (paridad) de cada registro del archivo corrupto con los bytes de verificación del archivo de verificación.

Recibe: **f1**: nombre y ruta del archivo corrupto (binario)
f2: nombre y ruta del archivo de verificadores (de texto)

Devuelve: **si éxito**: la cantidad de errores detectados
si error de archivo: -1

- c) Escriba un programa completo que solicite por teclado el nombre del archivo a verificar y el archivo de verificadores y utilice las funciones de los puntos 1a) y 1b) para determinar la cantidad de registros corruptos.

Nota: si realiza el examen en PC, cuenta con los siguientes archivos de trabajo:

datos.ddd: es el archivo corrupto que debe ser verificado. Es binario.

verif.vvv: es el archivo que contiene los bytes de verificación. Es de texto.



Apellido y Nombres	Legajo	# de Hojas	Profesor

2. La función de un *parser* es la “interpretación” de expresiones. Un ejemplo de esto es el software MathLab. Cuando Ud. ingresa una expresión en MathLab, la expresión ingresa como string. El *parser* la “interpreta”, analiza la sintaxis y, de ser correcta, la convierte a un formato en que puede procesarla.

Escribir un parser no es tarea fácil. Como 2do punto práctico, le pedimos que escriba una función o programa que solamente controle si hay consistencia entre paréntesis de una expresión ingresada por teclado (es decir, si los paréntesis que abren coinciden con los que cierran, y si están en orden correcto).

Para ello, la mejor estrategia consiste en utilizar una pila. Un paréntesis que abre se apila en la pila, y cuando cierra se retira de la misma. Si al final de la expresión la pila está vacía, los paréntesis están balanceados. En caso contrario, significa que la cantidad de paréntesis que abren no se corresponde con los que cierran, o que no se hallan en orden correcto.

Para implementar la pila, sugerimos que utilice un vector. Como alternativa, puede hacerlo con memoria dinámica.