Ejercicios prácticos sobre Programación en C

Programación Para Sistemas (PPS) 2021/22

Ejercicio 1.

Escribe un programa que lea enteros y los almacene en un array para imprimirlos en el orden inverso. Tu programa admitirá un máximo de M enteros (pongamos 1000). Si en la entrada estándar hay más de M enteros sólo se invertirán los M primeros. En el momento en el que haya algo diferente a un entero, el programar invertirá todos los enteros leídos hasta ese momento.

Para no tener que teclear números desde el teclado os sugerimos que pongáis los números en un fichero, digamos numeros.txt:

1
8
4000
2
23
fin
51
87
fin

Si suponemos que has llamado a tu programa invertir_enteros, esperamos que este sea el comportamiento: *./invertir_enteros < numeros.txt*

23
2
4000
8
1

Ejercicio 2.

Escribe un programa que lea de la entrada estándar dos matrices y las multiplique. Puedes asumir que las matrices están codificadas de la siguiente forma en la entrada estándar:

- Un entero positivo m que indica el número de filas de la primera matriz.
- Un entero positivo n que indica el número de columnas de la primera matriz (que coincide con el número de filas de la segunda).
- Un entero positivo p que indica el número de columnas de la segunda matriz.
- *m x n* floats: los n primeros son la primera fila de la primera matriz, los n segundos la segunda fila, etc.
- *n x p floats*: los p primeros son la primera fila de la segunda matriz, los p segundos la segunda fila, etc.

Se puede asumir que m, n y p están entre 1 y 1000. El resultado serán *m x p* floats donde los p primeros son la primera fila de la matriz multiplicación, los p segundos la segunda fila, etc.

Para no tener que teclear las matrices desde el teclado os sugerimos que las pongáis en un fichero, digamos *matrices.txt*:

3

2

3

1 2

-1 0

-3 -1

2 0 1

-5 2 3

Si suponemos que has llamado a tu programa multiplicar matrices, esperamos que este sea el comportamiento*:/multiplicar_matrices < matrices.txt*

-8 4 7

-2 0 -1

-1 -2 -6

Ejercicio 3.

Realice un programa (maxmin) que reciba como entrada el nombre de un fichero de texto que contiene un número real (float) por línea y que después de leerlo escriba por pantalla el valor máximo y el valor mínimo en una sola línea (formato %10.2f). Si el fichero está vacío debe escribir los valores 0.00 y 0.00. Suponer que los números del fichero son números reales válidos. Hay que comprobar que en la llamada se ha pasado un argumento de entrada, en caso contrario terminar sin imprimir nada y devolver al sistema operativo el número -1. Si el fichero no se puede leer también se devolverá el número -1 sin imprimir nada. En el resto de los casos, el número a devolver será el 0. Suponga que "fich.txt" contiene las siguientes 3 líneas:

10.2
12.55
15.0

Entonces, la salida de la llamada: *./maxmin fich.txt* deberán ser la siguiente línea:

15.00 10.20

Ejercicio 4.

Realice un programa "lineas_espaciadas.c" que reciba como entrada el nombre de un fichero texto y que, en salida estándar, muestre el contenido de dicho fichero a la vez que añade una línea en blanco después de cada salto de línea. Suponga que "fich.txt" contiene las siguientes 3 líneas que acaban en salto de línea:

Linea 1
Linea 2
Linea final.

Entonces, la salida de la llamada: *./lineas_espaciadas fich.txt* deberán ser las siguientes 6 líneas:

Linea 1

Linea 2

Linea final.

Si no se proporciona un argumento al programa, o bien dicho argumento no es el nombre de un fichero que se pueda leer, entonces el programa deberá mostrar en salida error la palabra "ERROR" junto con un salto de línea, y devolver al sistema operativo el número 1. En otro caso, el número a devolver será el 0.

Notas:

- El tamaño máximo de una línea es de 80 caracteres (incluyendo el posible salto de línea).

- Una línea en blanco es una línea que sólo contiene un salto de línea.

- La última línea de un fichero puede que contenga o no un salto de línea. El programa debe considerar adecuadamente ambos casos, y sólo añadir una línea en blanco final en el primer caso.

Ejercicio 5.

- Realice un programa "expandir_espacios.c" que expanda cada espacio de las líneas de la entrada estándar en un número de espacios indicado por argumento. La salida debe mostrarse en salida estándar. Suponga que "fich.txt" contiene las siguientes 2 líneas:

Linea primera y
linea final.

Entonces, la llamada: *cat fich.txt | ./expandir_espacios 3 l> salida.txt* deberá producir un fichero de salida "salida.txt" que contenga:

Linea primera y

linea final.

Si no se proporciona un argumento al programa, el programa deberá devolver al sistema operativo el número 1. Si el argumento existe, se puede asumir que se trata de la representación en caracteres de un número entero válido (es decir, no hace falta realizar control de errores al respecto), y al finalizar el programa se deberá devolver al sistema operativo el número 0.

Notas:

- El tamaño máximo de una línea es de 80 caracteres (incluyendo el posible salto de línea).

- Existe una función de librería de conversión simple (sin control de errores) de cadena de caracteres a número entero denominada "atoi" (ver: man 3 atoi).

- El programa no debe expandir los caracteres de salto de línea; sólo debe expandir el carácter espacio ' '.

Ejercicio 6.

Escribir un programa en un archivo de código fuente llamado *transformar.c* que lea del canal de entrada estándar una cadena de caracteres y la transforme en otra cadena de caracteres de manera que se reemplacen todas las secuencias de caracteres compuestos por un *backslash* (carácter '\') y una 't' o una 'n' por su equivalente, es decir, un *carácter tabulador* y un *carácter salto de línea*.

Se puede suponer que la cadena que introduce el usuario no tiene ningún blanco, es decir, no contiene: ni espacios en blanco (barra espaciadora), ni tabuladores ni saltos de línea. Siendo estrictos, se debería considerar que no habría que hacer ninguna sustitución en una secuencia como "\\n". Para evitar interpretar este caso en uno u otro sentido, también se puede suponer que no existen secuencias de letras como la indicada.

Se puede suponer que la longitud de la cadena leída es estrictamente menor que 80. Una vez hecha la transformación escriba el resultado en el canal de salida estándar en la forma. Por ejemplo, si la cadena original es:

Esta\tes\tla\tCadena\nResultado

Se debe escribir:

"Esta es la Cadena

Resultado".

Es decir, se debe mostrar como resultado (se recuerda: en un **único** printf): una comilla doble, la cadena resultado, otra comilla doble, un punto (final) y un salto de línea.

Nota 1: Se debe saber que los caracteres tabulador y salto de línea ocupan un único byte cuando se codifican en una cadena alfanumérica.

Nota 2: Es obligatorio formar la cadena alfanumérica. Dicho de otra manera, está prohibido generar los caracteres de uno a uno y mostrarlos sin incorporarlos a la cadena resultado. Es decir, solo se permite una única llamada a printf y no se permite usar ninguna variante de la función put.

Ejercicio 7.

En este ejercicio muestra el procesado de **argumentos en la línea de orden**, el manejo de **cadenas de caracteres** y de **ficheros tipo texto**. En concreto, el programa leerá un archivo de texto cuyo *path* se le suministra en la línea de orden o, en su defecto, de la entrada estándar. Dicho archivo contendrá, obligatoriamente, una **primera línea de cabecera** que define los campos de datos separados por un carácter ‘,’ (formato *csv*, *comma separated value*). La segunda y sucesivas líneas corresponde a los datos pudiendo existir líneas en blanco. En este caso, el programa las ignorará. Todos los datos se tratarán como caracteres. Finalmente, el programa mostrará en la salida estándar las líneas de datos etiquetadas de acuerdo a la cabecera. A continuación, se muestra un ejemplo de entrada y del resultado que el programa debe producir

```
ENTRADA (fichero o stdin)
=========================

title,album,duration,release,artist,type
Black Hole Sun,Superunknown,05:06,1994,Soundgarden,Rock
Smells Like Teen Spirit,Nevermind,05:01,1991,Nirvana,Grunge
Breed,Nevermind,03:03,1991,Nirvana,Grunge
Lithium,Nevermind,04:17,1991,Nirvana,Grunge
Once,Ten,03:51,1991,Pearl Jam,Grunge
Even Flow,Ten,04:53,1991,Pearl Jam,Grunge
Alive,Ten,05:40,1991,Pearl Jam,Grunge
Jeremy,Ten,05:18,1991,Pearl Jam,Grunge
Forty Six & 2,Aenima,06:04,1994,Tool,Metal Progresivo
Lateralus,Lateralus,09:24,2001,Tool,Metal Progresivo

SALIDA
======

title: Black Hole Sun; album: Superunknown; duration: 05:06; release: 1994; artist: Soundgarden; type: Rock
title: Smells Like Teen Spirit; album: Nevermind; duration: 05:01; release: 1991; artist: Nirvana; type: Grunge
title: Breed; album: Nevermind; duration: 03:03; release: 1991; artist: Nirvana; type: Grunge
title: Lithium; album: Nevermind; duration: 04:17; release: 1991; artist: Nirvana; type: Grunge
title: Once; album: Ten; duration: 03:51; release: 1991; artist: Pearl Jam; type: Grunge
title: Even Flow; album: Ten; duration: 04:53; release: 1991; artist: Pearl Jam; type: Grunge
title: Alive; album: Ten; duration: 05:40; release: 1991; artist: Pearl Jam; type: Grunge
title: Jeremy; album: Ten; duration: 05:18; release: 1991; artist: Pearl Jam; type: Grunge
title: Forty Six & 2; album: Aenima; duration: 06:04; release: 1994; artist: Tool; type: Metal Progresivo
title: Lateralus; album: Lateralus; duration: 09:24; release: 2001; artist: Tool; type: Metal Progresivo
```

Por último, se supone que la longitud máxima de las líneas en el fichero es de 2048 bytes y que el número máximo de campos es 15. En todo caso, esto debiera definirse a través de constantes de manera que se pudieran cambiar los límites sin tener que modificar código

El programa se codificará en tres archivos, dos fuentes .c (***main.c*** y ***parser.c***) y una cabecera .h (***parser.h***). Este último contendrá el siguiente código:

```
#define MaxLinea 2048
#define MaxFields 15

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int parser(FILE * file);
```

La función ***parser()*** es la que se encarga del procesado del fichero y de la escritura del resultado.

Ejercicio 8.

La ecuación dada por a * x^2 + b * x + c = 0, más conocida por *ecuación de segundo grado* tiene una solución bien conocida a través de una fórmula que permite obtener las raíces reales de la ecuación cuando se cumplen algunas condiciones. Sin embargo, es posible dar la solución para cualesquiera valores de a, b y c, excepto, naturalmente, que tanto a como b sean cero (sea cual sea el valor de c). Se pide:

Implementar una función cuya cabecera es:

int resolver(double a, double b, double c, double px1, double *px2);*

que resuelva la ecuación de segundo grado de acuerdo a las siguientes especificaciones:

1) Si las soluciones son reales se deben guardar en las variables apuntadas por px1 y px2

2) Si las soluciones son imaginarias se debe guardar la parte real de la solución en (la variable apuntada por) px1 y la parte imaginaria en (la variable apuntada por) px2. Como las dos soluciones imaginarias son conjugadas, es suficiente con guardar la solución con **parte imaginaria mayor que cero**.

3) Si la solución no existe se deben fijar las variables apuntadas por px1 y px2 al valor nan (*Not A Number*). Nota: en el compilador GCC se puede usar una macro: NAN que proporciona este *valor* especial.

4) Si existe una única solución (ecuación de primer grado), esta se debe guardar en px1 y debe guardarse el valor *nan* en px2.

La función debe devolver verdadero (uno, 1) cuando la solución sea compleja, es decir, en el caso 2 y falso (cero, 0) en cualquier otro caso.

Escribir el programa principal para que resuelva la ecuación, el programa debe leer los valores de a, b y c del canal de entrada estándar y escribir en el canal de salida la cadena "Compleja" si la solución es compleja y la cadena "No-Compleja" si no es. los dos valores de la solución separados por un espacio. Se recomienda comprobar el resultado, al menos, con los siguientes casos de prueba.

a	b	c
1.0	-2.0	1.0
-1.0	-2.0	1.0
3.0	-2.0	-8.0
-3.0	-2.0	-8.0
2.0	3.0	5.0
0.0	0.0	4.0
2.0	0.0	6.0

0.0 4.0 11.0

Tanto la función como el programa principal se deben escribir en un único archivo llamado **ec2g.c**

Ejercicio 9.

Escribir un programa (notas) que recibe como argumento de entrada el nombre de un fichero texto que contiene un listado de notas de alumnos de la clase. El número de líneas del fichero es desconocido. Cada línea tiene 3 campos: nombre (longitud máxima 50 caracteres), apellido (longitud máxima 100 caracteres) y nota (número entero). Tanto el nombre como el apellido no puede contener espacios en blanco. El programa tiene que:

- Usar la siguiente estructura para almacenar el nombre, apellido y nota de cada alumno que se lea del fichero y construir una **única** lista que se pueda recorrer:

```
typedef struct notas {
  char *nombre;
  char *apellido;
  int notas;
  struct notas *siguiente;
} lista_notas;
```

- El programa debe escribir por la salida estándar, primero los alumnos (nombre apellido nota) suspensos y a continuación los alumnos aprobados. Un alumno por línea y respetando el orden de aparición en el fichero.

Suponga que "fich.txt" contiene las siguientes 3 líneas:

Nombre1 Apellido1 8
Nombre2 Apellido2 2
Nombre3 Apellido3 6
Entonces, la salida de la llamada: *./notas fich.txt* deberán ser las siguientes líneas:

Nombre2 Apellido2 2
Nombre1 Apellido1 8
Nombre3 Apellido3 6

Cada vez que se haga una petición de memoria dinámica, se debe comprobar si la petición se realizó con éxito y en caso de que haya habido un problema, el programa acabará sin imprimir nada y devolviendo el código 71. Si no hay ningun problema en la ejecución devolvera el código 0 al finalizar. Se debe liberar toda la memoria dinámica solicitada antes de finalizar el programa.

Suponer que en la llamada al programa, siempre se pasa un fichero que existe, se puede leer y cumple el formato definido. No hace falta hacer comprobaciones del número de argumentos de entrada.

Ejercicio 10.

En este ejercicio muestra el uso de ***arrays***, ***structs*** y **definiciones de tipos (*typedef*)** en lenguaje C. Consiste en implementar una estructura de pila típica (*LIFO*) utilizando como base un array de tamaño predefinido y fijo. Los datos que contendrá la pila simulan información básica de alumnos y se definen en el archivo ***pilainfo.h*** mediante el siguiente *struct*:

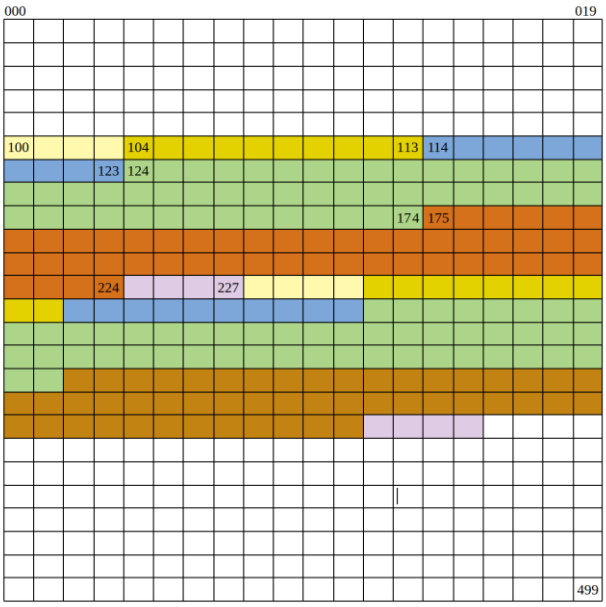
```
/*
 * Este tipo define la información que contiene la pila (el contenido)
 */
typedef struct pilainfo
{
    int id;
    char matricula[10];
    char dni[10];
    char nombre[50];
    char mail[50];
    float nota;
} tdata;
```

En el archivo ***pilavector.h*** se suministran las definiciones de tipos necesarias para construir la pila

```
/*
 * Este tipo es la propia pila, implementada como un array (el contenedor)
 * de MAXPILA (50) unidades tipo pdata (el contenido)
 */
typedef tdata tpila[MAXPILA];

/*
 * El tipo pilavector define la pila y el indice a la cima
 * Se accederá mediante los habituales servicios push/pop
 */
typedef struct pilavector {
  tdata datos[MAXPILA];
  int top;
} tpv;
```

La figura siguiente simula un volcado de memoria donde la pila ocuparía desde la celda nº 100 hasta la n.º 499 y que contiene dos elementos de datos, ocupando desde la celda n.º 100 hasta la n.º 355.



Se pide codificar en un archivo ***pilavector.c*** las funciones **void init(void)**, **void vaciar(void)**, **int llena(void)**, **int vacia(void)**, **int push**(tdata item), tdata **pop(void)** y **void display(void)**. Se suministran como material de apoyo los archivos ***pilainfo.h*** y ***pilavector.h***

La siguiente figura muestra la salida que se obtiene en pantalla al ejecutar un programa de test que realiza las siguientes operaciones:

1. Mostrar el tamaño que ocupan en memoria las estructuras de datos que usa el programa
2. Apilar dos alumnos previa inicialización explícita de la pila
3. Visualizar la pila
4. Comprobar si la pila está llena
5. Desapilar un dato y visualizar la pila
6. Vaciar la pila

```
Info consumo de memoria:
Tamaño pdata (struct pilainfo): 128 bytes
Tamaño pila == array pdata[50]: 6400 bytes
Tamaño pv (struct pilavector):  6404 bytes
== == == == == == == == == == == == == ==

Ini Test PilaVector
==================
==================

PilaVector.init()
================

PilaVector.push()
Contenido pila:
        ID: 2
        Matrícula: mat_a2
        Alumno: nombre_a2
        Mail: mail_a2
        Nota: 4.900000
        ID: 1
        Matrícula: mat_a1
        Alumno: nombre_a1
        Mail: mail_a1
        Nota: 3.000000

================

PilaVector.llena()
================

PilaVector.pop()
pop retorna ID: 2; matrícula: mat_a2; dni: dni_a2; nombre: nombre_a2; mail: mail_a2
Contenido pila:
        ID: 1
        Matrícula: mat_a1
        Alumno: nombre_a1
        Mail: mail_a1
        Nota: 3.000000

================

PilaVector.vaciar()
        PILA VACÍA

================
```

Ejercicio 11.

Escribe un programa que cree una matriz de *m filas x p columnas*, donde los valores de m y p serán dos enteros proporcionados como argumentos del programa. Se puede asumir que siempre se ejecuta el programa con sus 2 argumentos y que son dos números enteros válidos. Los elementos de la matriz (a_{ij}) serán de tipo *long int* y se calculan con las siguientes reglas:

- Fila 1 todos los elementos serán 1´s (a_{1j}=1 j=1….p)
- Columna 1 todos los elementos serán 1´s (a_{i1}=1 i=1….m)
- Restos de elementos a_{ij}= $a_{i-1j} + a_{ij-1}$ i=2….m j=2….p Si al calcular un elemento, se cumple que a_{ij} >1.e6 entonces a_{ij} =1

Una vez creada la matriz hay que escribirla en la salida estándar en formato matricial como se muestra en el ejemplo. Utilizar para imprimir cada elemento el formato *"%li\t"*. Si no se pudo crear la matriz por problemas de memoria, el programa acabará sin mostrar nada y devolviendo el código 71. Si no hay ningún problema en la ejecución, devolverá

el código 0. Al finalizar se debe liberar toda la memoria dinámica solicitada antes de finalizar el programa. Suponiendo que nuestro programa se llama matrizdinamica, la ejecución: ./matrizdinamica 4 5 produciría la siguiente salida:

```
1     1     1     1     1
1     2     3     4     5
1     3     6     10    15
1     4     10    20    35
```

Ejercicio 12.

Una cola o fila o por sus nombres en inglés: *queue* ó FIFO (*First in, First Out*) es una estructura abstracta de datos cuya interfaz (en sentido amplio) se podría definir en C mediante 5 funciones:

```
queue* create();

void add(queue*, type);

int is_empty(queue*);

type peek(queue*);

void poll(queue*);
```

Donde queue es el tipo de dato que sirve para implementar la cola, type es el tipo de dato de los elementos que se guardan en la cola y las funciones tienen cada una la funcionalidad habitual.

Además, se pueden hacer las siguientes suposiciones:

- Para create se entiende que crea una nueva cola (vacía) *dinámicamente* y es obligación del usuario de create hacer la **liberación** correspondiente.
- Para que no haya fugas de memoria se puede suponer que sólo se llama a free de la propia cola cuando la cola ya está vacía.
- Para peek y poll se puede suponer que la cola no está vacía, en otro caso el comportamiento será indeterminado.
- Se entiende que peek **no** modifica la cola.

Se desea implementar esta estructura abstracta mediante una cadena enlazada cuyos elementos contengan *arrays* de tipo de dato guardado en la cola. De esta manera se tendrán que realizar menos operaciones de reserva de memoria dinámica y los elementos en la cola serán contiguos la mayoría de las veces. Como tipo de dato a guardar en la cola se van a utilizar los números enteros (int).

Se pide: Escribir la implementación de las funciones `peek` y `poll` en un archivo *peek_poll.c*. Por supuesto, la implementación de la función `poll` debe ser tal que evite fugas de memoria. Escriba también un programa que le permita comprobar si la implementación realizada es correcta. Escriba ese programa de manera que lea de la entrada estándar:

- Un carácter **+** y un número entero para añadir elementos a la cola.
- Un carácter **-** para desencolar un número y escribirlo en la salida estándar.
- Un carácter **%** para desencolar todos los números en la cola y mostrarlos por pantalla en el orden en que se van quitando de la cola.
- Un carácter **f** para desencolar todos los elementos de la cola, escribirlos en la salida estándar, liberar la cola que se ha usado y terminar el programa.

Para realizar este ejercicio se proporciona el contenido de dos archivos: *queue.h* y *queue.c* con las declaraciones de los tipos necesarios y las implementaciones de las funciones que no se piden.

Contenido de queue.h:

```
#define DATA_SIZE 4

typedef struct node {
	int len;
	int data[DATA_SIZE];
	struct node* next;
} node_t;

typedef struct {
	struct node* first;
	struct node* last;
} queue;

queue* create();

void add(queue*, int);

int is_empty(queue*);

int peek(queue*);

void poll(queue*);
```

Contenido de queue.c:

```
#include <stdlib.h>

#include "queue.h"

queue* create() {
```

```
        queue* new_queue = malloc(sizeof(queue));
        new_queue->first = NULL;
        new_queue->last = NULL;
        return new_queue;
}
int is_empty(queue* q) {
        return q->first == NULL;
}
void add(queue* q, int number) {
        if ( is_empty(q) ) {
                q->first = malloc(sizeof(node_t));
                q->first->data[0] = number;
                q->first->len = 1;
                q->first->next = NULL;
                q->last = q->first;
        } else if ( q->last->len < DATA_SIZE ) {
                q->last->data[q->last->len] = number;
                ++(q->last->len);
        } else {
                node_t *nn = malloc(sizeof(node_t));
                nn->data[0] = number;
                nn->len = 1;
                nn->next = NULL;
                q->last->next = nn;
                q->last = nn;
        }
}
```