Examen escrito de Programación 1. Miércoles 31 de enero de 2018

- Se debe disponer sobre la mesa un documento de identificación con fotografía.
- Se debe comenzar a resolver cada uno de los problemas del examen en una hoja de papel diferente, para facilitar su calificación por distintos profesores. Escribir en cada hoja de papel nombre y apellidos y utilizar, en su caso, ambas caras de la hoja.
- Tiempo máximo para realizar este examen: 3.0 horas

Problema 1º (3.0 puntos)

En este problema se va a trabajar con ficheros binarios que almacenan una suma algebraica bajo el formato que describe la regla sintáctica `<ficheroSumaAlgebraica>` que se define a continuación.

```
<ficheroSumaAlgebraica> ::=  [ <dato> { <operador> <dato> } ]
<dato> ::=  double
<operador> ::=  char
```

Un <operador> solo puede estar representado por los datos de tipo char `'+'` (que denota una suma) y `'-'` (que denota una resta).

Se debe diseñar la función **`calcular`**`(nombre)` que se especifica a continuación, escribiendo su código `C++`. Se valorará la eficiencia del método algorítmico aplicado y la legibilidad de su código. En la solución presentada, se admite que en la especificación de la función pedida se omitan las líneas que ilustran su comportamiento mediante un ejemplo.

```
/*
 * Pre: <nombre> almacena una cadena de caracteres con el nombre de un fichero binario cuyos
 *      datos definen una suma algebraica según la regla sintáctica <ficheroSumaAlgebraica>
 * Post: Devuelve el resultado de calcular la suma algebraica descrita por el contenido
 *       del fichero <nombre>
 * Ejemplo: Si el contenido del fichero <nombre> es el siguiente :
 *              [ <1.5> <'+'> <13.85> <'+'> <4.15> <'-'> <8.2> <'+'> <6.53> ]
 *          entonces calcular(nombre) = 17.83
 *          ya que la suma algebraica 1.5 + 13.85 + 4.15 - 8.2 + 6.53 es igual a 17.83
 */
double calcular (const char nombre[]);
```

Problema 2º (3.5 puntos)

En este problema se va a trabajar con cadenas de caracteres que almacenan una frase bajo los formatos que describen las reglas sintácticas <frase> y <fraseSimplificada> que se definen a continuación.

```
<frase> ::= [ <seps> ] [ <palabra> { <seps> <palabra> } ] [ <seps> ] <nulo>
<fraseSimplificada> ::= [ <palabra> { <blanco> <palabra> } ] <nulo>
<palabra> ::= <letra> { <letra> }
<letra> ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' |
            'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'x' | 'y' | 'z' |
            'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' |
            'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'X' | 'Y' | 'Z'
<seps> ::= <blanco> { <blanco> }
<blanco> ::= '␣'
<nulo> ::= '\0'
```

Se debe realizar un diseño descendente de la función **simplificar**(frase,simplificada) que se especifica a continuación, escribiendo su código C++, junto al código de, al menos, un par de funciones auxiliares que faciliten el diseño de la primera. Todas las funciones que integren la solución vendrán precedidas por sus correspondientes especificaciones **pre**/**post**. En la solución presentada, se admite que en la especificación de la función **simplificar**(frase,simplificada) se omitan las líneas que ilustran su comportamiento mediante ejemplos.

En el código de ninguna de las funciones que integran la solución no se permite el uso de ninguna de las funciones predefinidadas en las bibliotecas C++.

```
/*
 * Pre: <frase> almacena una cadena de caracteres con una secuencia de palabras atendiendo
 *      a la sintaxis descrita por la regla <frase>
 * Post: <simplificada> almacena una cadena de caracteres con la misma secuencia de palabras
 *      que <frase> con la diferencia de que atiende a la sintaxis descrita por la regla
 *      <fraseSimplificada>
 *
 * Ejemplos que muestran los valores de <frase> y de <simplificada> tras ejecutar
 * la invocación simplificar (frase, simplificada):
 *
 *       frase                                  simplificada
 *       =====                                  ============
 *       ""                                     ""
 *       "  "                                   ""
 *       "hola  "                               "hola"
 *       "Uno dos tres y cuatro  "              "Uno dos tres y cuatro"
 *       "  Uno dos tres y cuatro"              "Uno dos tres y cuatro"
 *       " Uno dos tres y cuatro  "             "Uno dos tres y cuatro"
 */
void simplificar (const char frase [], char simplificada []);
```

Problema 3º (3.5 puntos)

En este problema se va a trabajar con matrices o tablas bidimensionales que almacenan **NF**×**NC** datos enteros de tipo **int**. El número de filas y el de columnas de las matrices está definido por las constantes **NF** y **NC**, respectivamente, que han sido definidas en el programa con antelación a la función que se ha de diseñar.

```
// Número de filas (NF) y número de columnas (NC) de las matrices con las que se va a trabajar
const int NF = ...;          // El valor de NF es mayor que 0
const int NC = ...;          // El valor de NC es mayor que 0
```

Se debe realizar un diseño desdendente de la función **distribuir**(m,limite) que se especifica a continuación, escribiendo su código C++, junto al código de una o más funciones auxiliares que faciliten el diseño de la primera.

En este problema, que es un problema de distribución de los datos de una matriz, se valorará la eficiencia del método algorítmico aplicado y la legibilidad del código. La función pedida se presentará completa con su especificación y su código. Del mismo modo se presentará la función o funciones auxiliares en las que pudiera apoyarse su diseño.

```
/*
 * Pre: NF > 0 y NC > 0
 * Post: Los NFxNC elementos de la matriz <m> son una permutación de sus NFxNC elementos iniciales
 *       (los elementos al ser invocada la función). Si se recorre la matriz <m> por filas (partiendo
 *       de la fila 0 y acabando en la fila NF−1) y se recorre cada fila por columnas (partiendo de
 *       la columna 0 y acabando en la columna NC−1), los primeros elementos recorridos tienen valor
 *       menor que <limite>. En el momento en que un elemento tenga un valor igual o mayor que
 *       <limite>, los restantes elementos recorridos de la matriz <m> tendrán todos ellos también
 *       un valor igual o mayor que <limite>
 */
void distribuir (int m[][NC], const int limite );
```

Supongamos que las constantes **NF** y **NC** han sido definidas con los siguientes valores:

```
// Número de filas (NF) y número de columnas (NC) de las matrices con las que se va a trabajar
const int NF = 3;          // El valor de NF ha es mayor que 0
const int NC = 5;          // El valor de NC es mayor que 0
```

Si el valor inicial de la tabla bidimensional T de dimensión $NF \times NC$, es decir, de dimensión 3×5, es el siguiente:

-2	-9	0	-13	9
12	-6	-3	0	-12
4	-2	8	-20	45

Después de invocar la ejecución de **distribuir**(T,0) la tabla bidimensional T puede presentar como contenido cualquiera de las permutaciones de sus datos iniciales que satisfaga las propiedades descritas en la postcondición de la función. Por ejemplo el contenido que se muestra a continuación en el que todos los elementos presentes inicialmente en T cuyo valor era inferior al del parámetro **limite** (valor 0 en este caso) preceden ahora a cualquier otro elemento inicial de T con valor igual o mayor que el del parámetro **limite**.

-2	-9	-20	-13	-2
-12	-6	-3	0	12
4	9	8	0	45

Una solución del problema 1º

```
/*
 * Reglas sintácticas :
 *     <ficheroSumaAlgebraica> ::= [ <dato> { <operador> <dato> } ]
 *     <dato> ::= double
 *     <operador> ::= char
 *
 * Un <operador> solo puede estar representado por los datos de tipo char '+' y '-'
 */

/*
 * Pre: <nombre> almacena una cadena de caracteres con el nombre de un fichero binario
 *       cuyos datos están organizados según la regla sintáctica <ficheroSumaAlgebraica>
 * Post: Devuelve el resultado de calcular la suma algebraica descrita por el
 *        contenido del fichero <nombre>
 * Ejemplo: Si el contenido del fichero <nombre> es el siguiente :
 *               [ <1.5> <'+'> <13.85> <'+'> <4.15> <'-'> <8.2> <'+'> <6.53> ]
 *            entonces calcular(nombre) = 17.88
 *            ya que la suma algebraica 1.5 + 13.85 + 4.15 - 8.2 + 6.53 es igual a 17.83
 */
double calcular (const char nombre[]) {
    // Antes de leer ningún dato <resultado> parte de un valor igual a 0
    double resultado = 0.0;
    ifstream f (nombre, ios :: binary );
    if (f.is_open ()) {
        double dato;
        // Lee el primer dato del fichero asociado a <f>
        f.read( reinterpret_cast <char *>(&dato), sizeof(double));
        // Asigna a <resultado> el valor de <dato>
        resultado = dato;
        while (!f.eof ()) {
            char op;
            // Intenta leer un operador del fichero asociado a <f>
            f.read( reinterpret_cast <char *>(&op), sizeof(char));
            // Intenta leer un nuevo dato del fichero asociado a <f>
            f.read( reinterpret_cast <char *>(&dato), sizeof(double));
            if (!f.eof ()) {
                if (op == '+') {
                    // Incrementa <resultado> con el valor de <dato>
                    resultado = resultado + dato;
                }
                else {
                    // Decrementa <resultado> con el valor de <dato>
                    resultado = resultado - dato;
                }
            }
        }
        f.close ();
    }
    // Devuelve el valor de <resultado>
    return resultado ;
}
```

Una solución del problema 2º

```
/*
 * Reglas sintácticas :
 *     <frase> ::= [ <seps> ] [ <palabra> { <seps> <palabra> } ] [ <seps> ] <nulo>
 *     <fraseSimplificada> ::= [ <palabra> { <blanco> <palabra> } ] <nulo>
 *     <palabra> ::= <letra> { <letra> }
 *     <letra> ::= 'a' | 'b' | ... | 'y' | 'z' | 'A' | 'B' | ... | 'Y' | 'Z'
 *     <seps> ::= <blanco> { <blanco> }
 *     <blanco> ::= ' '
 *     <nulo> ::= '\0'
 */

/* Constante simbólica para el carácter espacio en blanco */
const char BLANCO = '␣';

/* Constante simbólica para el carácter nulo */
const char NULO = '\0';

/*
 * Pre:  «cadena» almacena una cadena de caracteres con una secuencia de palabras atendiendo
 *       a la sintaxis descrita por la regla <frase> y 0 <= i <= longitud(cadena).
 * Post: «i» apunta a la primera letra de «cadena» apartir de la
 *       posición que tenía «i» cuando se invocó esta función o al
 *       carácter nulo de «cadena».
 */
void saltarEspacios (const char cadena[], int& i) {
    while (cadena[i] == BLANCO) {
        // No hace falta comprobar cadena[i] != NULO, ya que NULO != BLANCO
        i++;
    }
    // cadena[i] != BLANCO, lo que equivale a que cadena[i] cumple con
    // la regla sintáctica <letra> o con la de <nulo>.
}

/*
 * Pre:  «frase» almacena una cadena de caracteres con una secuencia de
 *       palabras atendiendo a la sintaxis descrita por la regla
 *       <frase>, 0 <= indFrase <= longitud(frase ),
 *       indSimplificada >= 0.
 * Post: Ha copiado la palabra existente en «frase» a partir de la
 *       posición igual al valor inicial de «indFrase» en la cadena
 *       simplificada , a partir de la posición inicial de
 *       «indSimplificada» . «indFrase» e «indSimplificada» se han
 *       incrementado en el número de letras que tenía la palabra
 *       existente en «frase» a partir de la posición igual al valor
 *       inicial de «indFrase».
 */
void copiarPalabra (const char frase [], int& indFrase,
                    char simplificada [], int& indSimplificada ) {
    while ( frase [indFrase ] != BLANCO && frase[indFrase] != NULO) {
        // Dada la regla sintáctica <frase>, la condición anterior
        // es equivalente a decir que frase[indFrase] cumple con la
        // regla sintáctica <letra>.
        simplificada [ indSimplificada ] = frase [indFrase ];
        indFrase++;
        indSimplificada ++;
    }
    // frase[indFrase] == BLANCO || frase[indFrase] == NULO
}
```

```
/*
 * Pre:  «frase» almacena una cadena de caracteres con una secuencia de
 *       palabras atendiendo a la sintaxis descrita por la regla
 *       <frase>.
 * Post: «simplificada» almacena una cadena de caracteres con la misma
 *       secuencia de palabras que «frase» con la diferencia de que
 *       atiende a la sintaxis descrita por la regla
 *       <fraseSimplificada>.
 */
void simplificar (const char frase [], char simplificada []) {
    // Índices para recorrer «frase» y «simplificada», respectivamente
    int indFrase = 0, indSimplificada = 0;

    // Salto de los espacios en blanco que pueda haber al principio de «frase»
    saltarEspacios ( frase , indFrase );

    while ( frase [ indFrase ] != NULO) {
        // En frase[indFrase] comienza una <palabra>. Se copia a
        // «simplificada» y se actualizan los índices.
        copiarPalabra ( frase , indFrase , simplificada , indSimplificada );

        // Salto de los espacios en blanco que pueda haber tras la
        // última <palabra> tratada.
        saltarEspacios ( frase , indFrase );

        // frase[indFrase] es NULO o <letra>. Solo si es <letra>, hay
        // otra <palabra> más en «frase», por lo que se añade un BLANCO
        // en «simplificada».
        if ( frase [ indFrase ] != NULO) {
            simplificada [ indSimplificada ] = '␣';
            indSimplificada ++;
        }
    }
    // frase[indFrase] == NULO

    // Se pone el carácter NULO en «simplificada».
    simplificada [ indSimplificada ] = NULO;
}
```

Una solución del problema 3º

```
const int NF = 4;          // Número de filas de las matrices con las que se va a trabajar
const int NC = 3;          // Número de columnas de las matrices con las que se va a trabajar

/*
 * Pre: <fila> y <columna> representan los índices de la fila y columna
 *      de un elemento de una matriz de dimensión NF x NC
 * Post: <fila> y <columna> representan los índices de la fila y columna
 *       del siguiente elemento de la matriz de dimensión NF x NC citada en la
 *       precondición si la matriz se recorre por filas (partiendo de la fila 0
 *       y acabando en la fila NF-1) y, cada fila se recorre por columnas (partiendo
 *       de la columna 0 y acabando en la columna NC-1)
 */
void avanzar (int& fila, int& columna) {
    if (columna < NC - 1) {
        columna = columna + 1;
    }
    else {
        fila = fila + 1;
        columna = 0;
    }
}

/*
 * Pre: <fila> y <columna> representan los índices de la fila y columna
 *      de un elemento de una matriz de dimensión NF x NC
 * Post: <fila> y <columna> representan los índices de la fila y columna
 *       del siguiente elemento de la matriz de dimensión NF x NC citada en la
 *       precondición si la matriz se recorre por filas (partiendo de la fila NF-1
 *       y acabando en la fila 0) y, cada fila se recorre por columnas (partiendo de
 *       la columna NC-1 y acabando en la columna 0)
 */
void retroceder (int& fila, int& columna) {
    if (columna > 0) {
        columna = columna - 1;
    }
    else {
        fila = fila - 1;
        columna = NC - 1;
    }
}
```

```
/*
 * Pre: NF > 0 y NC > 0
 * Post: Los NFxNC elementos de la matriz <m> son una permutación de sus NFxNC elementos iniciales
 *       (los elementos al ser invocada la función). Si se recorre la matriz <m> por filas (partiendo
 *       de la fila 0 y acabando en la fila NF−1) y se recorre cada fila por columnas (partiendo de
 *       la columna 0 y acabando en la columna NC−1), los primeros elementos recorridos tienen valor
 *       menor que <limite>. En el momento en que un elemento tenga un valor igual o mayor que
 *       <limite>, los restantes elementos recorridos de la matriz <m> tendrán todos ellos también
 *       un valor igual o mayor que <limite>
 */
void distribuir (int m[][NC], const int limite) {
    int infFila = 0, infCol = 0;
    int supFila = NF − 1, supCol = NC − 1;
    while ( infFila < supFila || ( infFila == supFila && infCol < supCol)) {
        if (m[infFila][infCol] < limite) {
            // Actualiza los valores de los índices <infFila> e <infCol>
            avanzar( infFila , infCol );
        }
        else if (m[supFila][supCol] >= limite) {
            // Actualiza los valores de los índices <supFila> y <supCol>
            retroceder (supFila , supCol);
        }
        else {
            // Permuta m[infFila][infCol] y m[supFila][supCol];
            int aux = m[infFila][infCol];
            m[infFila][infCol] = m[supFila][supCol];
            m[supFila][supCol] = aux;
            // Actualiza los valores de los índices <infFila>, <infCol>,
            // <supFila> y <supCol>
            avanzar( infFila , infCol );
            retroceder (supFila , supCol);
        }
    }
}
```