1. Especificación del TAD

```
1  espec PilaDelimitadores
2  usa caracteres, booleanos
3  genéro PilaDelim
4  { Los valores del TAD PilaDelim representan una pila de caracteres
5   delimitadores de apertura, que sigue un comportamiento LIFO (Last
6   In, First Out). Se pueden almacenar caracteres como '(', '{', '['
7   para realizar validaciones de apertura y cierre de delimitadores
8   en una expresión. Los valores de PilaDelim disponen de operaciones
9   que permiten apilar caracteres, desapilar el último carácter
10  apilado, consultar el tope de la pila y verificar si la pila está
11  vacía. }
12 operaciones
13     inicializar: -> PilaDelim { operación generadora }
14     { Devuelve una pila vacía, sin ningún carácter delimitador }
15
16     apilar: PilaDelim p, carácter c -> PilaDelim { generadora }
17     { Devuelve la pila resultante de añadir el carácter
18     delimitador c a la pila p. Si c no es un carácter delimitador
19     o la pila está llena, devuelve la pila p sin modificar }
20
21     parcial desapilar: PilaDelim p -> PilaDelim { modificadora }
22     { Devuelve la pila resultante de eliminar de p el último
23     carácter delimitador que fue apilado.
24     Parcial: la operación no está definida si estáVacía?(p). }
25
26     parcial cima: PilaDelim p -> carácter { observadora }
27     { Devuelve el carácter que está en la parte superior de la
28     pila p.
29     Parcial: la operación no está definida si estáVacía?(p). }
30
31     estáVacía?: PilaDelim p -> booleano { observadora }
32     { Devuelve verdad si la pila p está vacía (es decir, no
33     contiene ningún carácter delimitador), falso en caso
34     contrario. }
35 fespec
```

Resumen de consejos para escribir una buena especificación

- **Descripción general del TAD**:
  - Indica qué representan los valores del TAD.
  - Define qué tipo de datos representa el TAD y qué información almacena.
- **Descripción de entidades** (elementos dentro del TAD):
  - Detalla cada entidad, explicando qué información caracteriza a cada entidad dentro del TAD.
  - Identifica las claves o criterios que diferencian los elementos, especialmente si se trata de diccionarios o colecciones sin elementos repetidos.
- **Restricciones y operaciones**:
  - Describe las restricciones: si existen limitaciones o condiciones especiales que deben cumplirse, indícalas claramente.
  - Define las operaciones, proporcionando una breve descripción de cada operación disponible en el TAD.
  - Explica qué hace cada operación, sin entrar en detalles sobre cómo se implementa.
- **Especificación de operaciones**:
  - Comportamiento de las operaciones:
    - *Dominio*: Define la información que cada operación necesita como entrada.
    - *Rango*: Describe el resultado que produce la operación.
  - Operaciones parciales: Indica claramente las situaciones en las que una operación parcial no está definida o no puede producir un resultado.
- **Operaciones auxiliares**:
  - Define operaciones auxiliares si es necesario: a veces es útil incluir operaciones auxiliares que pueden ser utilizadas por otras operaciones.
- **Evita detalles de implementación**:
  - No incluyas detalles de implementación: una especificación debe centrarse en el comportamiento y las propiedades del TAD, no en cómo se implementa.
  - Evita describir métodos específicos de implementación o detalles técnicos.

2. Implementación del TAD

2.1. Pseudocódigo

```
1  módulo PilaDelimitadores
2  exporta
3      constante
4          TAM_MAX = 1000
5      tipo pilaDelim
6      { Pila de caracteres delimitadores de apertura ('(', '{', '['). 
7       Implementación limitada a pilas con un tamaño máximo de
8       TAM_MAX elementos }
9
10     procedimiento inicializar(sal p: pilaDelim)
11     { Crea una pila vacía (es decir, que no contiene ningún
12     carácter delimitador de apertura) }
13
14     procedimiento apilar(e/s p: pilaDelim; ent c: carácter;
15                          sal error: booleano)
16     { Si c es un carácter delimitador de apertura, y p no está
17     llena, entonces devuelve en el mismo parámetro p la pila
18     resultante de añadir c a p y error = falso. En cualquier otro
19     caso, la pila p no se modifica y error = verdad }
20
21     procedimiento desapilar(e/s p: pilaDelim; sal c: carácter)
22     { Pre: p es no vacía }
23     { Devuelve la pila resultante de eliminar el último elemento
24     que fue apilado, dejándolo en c }
25
26     función cima(p: pilaDelim) devuelve carácter
27     { Pre: p es no vacía }
28     { Devuelve el último elemento apilado en p }
29
30     función estáVacía?(p: pilaDelim) devuelve booleano
31     { Devuelve verdad si y sólo si p no tiene elementos }
32
33 implementación
34     tipos
35         índice    = 0..TAM_MAX  {indica la posición en la pila}
36         pilaDelim = registro
37                         datos: vector[1..TAM_MAX] de carácter;
38                         tope: índice;
39                     freg
40
```

```
41
42      procedimiento inicializar(sal p: pilaDelim)
43      { Crea una pila vacía (es decir, que no contiene ningún
44      carácter delimitador de apertura }
45      principio
46          p.tope = 0;
47      fin
48
49      función estáLlena?(p: pilaDelim) devuelve booleano
50      { Devuelve verdad si la pila p está llena, falso en caso
51      contrario }
52      principio
53          devuelve (p.tope = TAM_MAX)
54      fin
55
56      función esDelimitador?(c: carácter) devuelve booleano
57      { Devuelve verdad si c es un delimitador de apertura, falso
58      en caso contrario }
59      principio
60          devuelve (c = '(' or c = '[' or c = '{')
61      fin
62
63      procedimiento apilar(e/s p: pilaDelim; ent c: carácter;
64                           sal error: booleano)
65      { Si c es un carácter delimitador de apertura, y p no está
66      llena, entonces devuelve en el mismo parámetro p la pila
67      resultante de añadir c a p y error = falso. En cualquier otro
68      caso, la pila p no se modifica y error = verdad }
69      principio
70          error := verdad;
71          si not estáLlena?(p) and esDelimitador(c) entonces
72              p.tope := p.tope + 1;
73              p.datos[p.tope] := c;
74              error := falso;
75          fsi
76      fin
77
```

```
78
79      { Versión NO ROBUSTA }
80      procedimiento desapilar(e/s p: pilaDelim; sal c: carácter)
81      { Pre: p es no vacía }
82      { Devuelve la pila resultante de eliminar el último elemento
83      que fue apilado, dejándolo en c }
84      principio
85              c := p.datos[p.tope];  {Obtener el valor en la cima}
86          p.tope := p.tope - 1  {Decrementar el tope}
87      fin
88
89      { Versión ROBUSTA }
90      procedimiento desapilar(e/s p: pilaDelim; sal c: carácter;
91                                  sal error: booleano)
92      { Si p es no vacía, devuelve la pila resultante de eliminar
93      el último elemento que fue apilado, dejándolo en c y
94      error = falso. Si p es vacía, la deja igual y error = verdad }

95      principio
96          error := verdad;
97          si not estáVacía?(p) entonces
98                    c := p.datos[p.tope];
99              p.tope := p.tope - 1;
100              error := falso;
101         fsi
102     fin
103
104
```

```
105
106     { Versión NO ROBUSTA }
107     función cima(p: pilaDelim) devuelve carácter
108     { Pre: p es no vacía }
109     { Devuelve el último elemento apilado en p }
110     principio
111         devuelve p.datos[p.tope]
112     fsi
113
114     { Versión ROBUSTA }
115     procedimiento cima(ent p: pilaDelim; sal c: caracter;
116                           sal error: booleano)
117     { Si p es vacía, error toma el valor verdad y se deja c
118     indefinido. Si p no es vacía, error toma el valor falso y c
119     toma el valor de la cima de p (es decir, c = cima(p)) }
120     principio
121         error := verdad;
122         si not estáVacía?(p) entonces
123                 c := p.datos[p.tope];
124             error := falso;
125         fsi
126     fin
127
128     función estáVacía?(p: pilaDelim) devuelve booleano
129     { Devuelve verdad si y sólo si p no tiene elementos }
130     principio
131           devuelve (p.tope = 0)
132     fin
133 fin
```


3. Ejemplo de uso del módulo

```
procedimiento balanceada
{Lee una secuencia de caracteres de un fichero correspondiente a
una expresión matemática, donde se hace uso de delimitadores
(paréntesis, corchetes y llaves) y escribe por pantalla si la
expresión leída está o no bien balanceada.}
importa pilaDelimitadores, cadenas, ficheros

    { Pre: c es un carácter delimitador de apertura }
    función cierre(c: carácter) devuelve carácter
    principio
        si c = '(' entonces
            devuelve ')'
        sino_si c = '[' entonces
            devuelve ']'
        sino_si c = '{' entonces
            devuelve '}'
        fsi
    fin

variables
    f: fichero de caracteres;
    nombre: cadena;
    p: pilaDelim;
    dato: carácter;
    c: carácter;
    error: booleano := falso;
    balanceada: booleano := verdad
principio
    escribir("Nombre del fichero: ");
    leer(nombre);
    asociar(f, nombre);
    iniciarlectura(f);
    inicializar(p);
    mientrasQue (not finFichero(f) and
                    not error and balanceada) hacer
        leer(f, dato);
        si (dato = '(' or dato = '[' or dato = '{') entonces
            apilar(p, dato, error);
        sino_si (dato = ')' or dato = ']' or dato = '}') entonces
            desapilar(p, c, error);
            balanceada := (dato = cierre(c))
        fsi
    fmq;
    disociar(f);
    si error
        escribir("Error por saturación de la capacidad de la pila utilizada.")
    sino
        escribir("La expresión leída está ")
        si not balanceada entonces
            escribir("IN")
        fsi
        escribir("CORRECTAMENTE balanceada")
    fsi
fin
```


4. Resumen de pasos

1. **Análisis de la especificación**
   - Revisa la especificación formal para entender claramente los objetivos y el comportamiento esperado del sistema o módulo.
   - Identifica las operaciones y propiedades principales que deben implementarse, así como las precondiciones, postcondiciones y posibles restricciones (por ejemplo, límites de tamaño, tipos de datos, etcétera).
2. **Diseño de la representación interna**
   - Define cómo se representarán los datos internamente. Selecciona las estructuras de datos que soporten las operaciones descritas en la especificación de manera eficiente.
3. **Implementación de operaciones**
   - Para cada operación especificada, escribe la implementación concreta basada en las estructuras de datos elegidas.
   - Asegúrate de que cada operación respete las propiedades establecidas en la especificación.
4. **Manejo de errores**
   - Si se especifican versiones robustas, implementa mecanismos para el manejo adecuado de errores, como intentar realizar operaciones inválidas.
   - Asegúrate de que las funciones y procedimientos devuelvan indicaciones claras cuando ocurra un error, y no permitan que el programa falle inesperadamente. Revisa la especificación si es necesario.
5. **Pruebas y validación**
   - Realiza pruebas exhaustivas para validar que la implementación cumple con la especificación. Asegúrate de que todos los casos límite estén cubiertos.
   - Verifica que todas las operaciones se comporten como se describe en la especificación, incluyendo casos de éxito y de error.
6. **Optimización y mejora**
   - Una vez validadas las funcionalidades, vuelve a pensar si tu estructura planteada e implementada es la más eficiente. Optimiza la implementación si es necesario, buscando mejorar la eficiencia en tiempo y en espacio.
   - Asegúrate de que la implementación siga siendo correcta tras cualquier ajuste.
7. **Documentación**
   - Documenta adecuadamente cada parte de la implementación para que otros desarrolladores puedan entender fácilmente cómo funciona y cómo usar el módulo o sistema.

5. Implementación en C++

5.1. Fichero «pila_delim.hpp»

```
1  #ifndef _PILADELIM_HPP
2  #define _PILADELIM_HPP
3
4  // Constantes y tipos previos
5  const unsigned TAM_MAX = 1000;
6
7  // Interfaz del TAD PilaDelim. Pre-declaraciones:
8
9  /* Los valores del TAD PilaDelim representan una pila de caracteres
10  * delimitadores de apertura, que sigue un comportamiento LIFO (Last
11  * In, First Out). Se pueden almacenar caracteres como ('(', '{', '['*)
12  * para realizar validaciones de apertura y cierre de delimitadores
13  * en una expresión. Los valores de PilaDelim disponen de operaciones
14  * que permiten apilar caracteres, desapilar el último carácter,
15  * consultar el tope de la pila y verificar si la pila está vacía.
16  */
17 struct PilaDelim;
18
19 /* Crea una pila vacía (es decir, que no contiene ningún
20  * carácter delimitador de apertura) */
21 void inicializar(PilaDelim& p);
22
23 /* Si c es un carácter delimitador de apertura, y p no está
24  * llena, entonces devuelve en el mismo parámetro p la pila
25  * resultante de añadir c a p y error = falso. En cualquier otro
26  * caso, la pila p no se modifica y error = verdad */
27 void apilar(PilaDelim& p, char c, bool& error);
28
29 /* Si p no es vacía, devuelve la pila resultante de eliminar
30  * el último elemento que fue apilado, dejándolo en c, y error = falso.
31  * Si la pila es vacía, error = verdad */
32 void desapilar(PilaDelim& p, char& c, bool& error);
33
34 /* Si p no es vacía, devuelve en c el último elemento apilado en p y
35  * error = falso. En caso contrario, error = verdad y c está indefinido */
36 void cima(const PilaDelim& p, char& c, bool& error);
37
38 /* Devuelve verdad si y sólo si p no tiene elementos */
39 bool estaVacia(const PilaDelim& p);
40
41 // Declaración
42
43 struct PilaDelim {
44     friend void inicializar(PilaDelim& p);
45     friend void apilar(PilaDelim& p, char c, bool& error);
46     friend void desapilar(PilaDelim& p, char& c, bool& error);
47     friend void cima(const PilaDelim& p, char& c, bool& error);
48     friend bool estaVacia(const PilaDelim& p);
49     friend bool estaLlena(const PilaDelim& p);
50
51     private: // Representación interna de los valores del TAD
52         char datos[TAM_MAX];
53         int tope;
54 };
55
56 #endif
```

5.2. Fichero «pila_delim.cpp»

```
1  #include "pila_delim.hpp"
2
3  /* Crea una pila vacía (es decir, que no contiene ningún
4   * carácter delimitador de apertura) */
5  void inicializar(PilaDelim& p) {
6      p.tope = -1; // en C, los vectores comienzan obligatoriamente en 0
7      // el pseudo-código planteado anteriormente variará ligeramente
8  }
9
10 /* Devuelve verdad si y sólo si p ha alcanzado su máximo de capacidad */
11 bool estaLlena(const PilaDelim& p) {
12     return p.tope == (TAM_MAX - 1);
13 }
14
15 /* Pre: c es un carácter delimitador de apertura ('(', '[', '{')
16  * Devuelve verdad si y sólo si c es un carácter delimitador de apertura */
17 bool esDelimitador(char c) {
18     return (c == '(' || c == '[' || c == '{');
19 }
20
21
22 /* Si c es un carácter delimitador de apertura, y p no está
23  * llena, entonces devuelve en el mismo parámetro p la pila
24  * resultante de añadir c a p y error = falso. En cualquier otro
25  * caso, la pila p no se modifica y error = verdad */
26 void apilar(PilaDelim& p, char c, bool& error) {
27     error = true;
28     if (!estaLlena(p) && esDelimitador(c)) {
29         p.tope = p.tope + 1;
30         p.datos[p.tope] = c;
31         error = false;
32     }
33 }
34
35 /* Si p no es vacía, devuelve la pila resultante de eliminar
36  * el último elemento que fue apilado, dejándolo en c, y error = falso.
37  * Si la pila es vacía, error = verdad */
38 void desapilar(PilaDelim& p, char& c, bool& error) {
39     error = true;
40     if (!estaVacia(p)) {
41         c = p.datos[p.tope];
42         p.tope = p.tope - 1;
43         error = false;
44     }
45 }
46
47 /* Si p no es vacía, devuelve en c el último elemento apilado en p y
48  * error = falso. En caso contrario, error = verdad y c está indefinido */
49 void cima(const PilaDelim& p, char& c, bool& error) {
50     error = true;
51     if (!estaVacia(p)) {
52         c = p.datos[p.tope];
53         error = false;
54     }
55 }
56
```

```
57 /* Devuelve verdad si y sólo si p no tiene elementos */
58 bool estaVacia(const PilaDelim& p) {
59     return (p.tope == -1);
60 }
```

5.3. Fichero «main.cpp» (uso del TAD)

```
1  #include <iostream>
2  #include <string>
3  #include "pila_delim.hpp"
4  using namespace std;
5
6  /* Devuelve el carácter delimitador de cierre de c */
7  char cierre(char c) {
8      if (c == '(') {
9          return ')';
10     } else if (c == '[') {
11         return ']';
12     } else if (c == '{') {
13         return '}';
14     } else{
15         cerr << "Carácter " << c << " inesperado!";
16         return -1;
17     }
18 }
19
20 /* Comprueba si la cadena s está balanceada usando PilaDelim
21  * Ojo: se asume que la cadena vacía es una expresión balanceada */
22 bool balanceada(string s) {
23     PilaDelim p;
24     bool error = false,
25          estaBalanceada = true;
26     unsigned len = s.length();
27
28     inicializar(p);
29
30     for (unsigned i = 0; i < len && !error && estaBalanceada; i++){
31         char c = s[i], c2;
32         if (c == '(' || c == '[' || c == '{') {
33             apilar(p, c, error);
34         } else if (c == ')' || c == ']' || c == '}') {
35             desapilar(p, c2, error);
36             estaBalanceada = (cierre(c2) == c);
37         }
38     }
39     return estaBalanceada && estaVacia(p);
40 }
41
```

```
42 /*
43  * Programa principal que solicita una expresión al usuario y comprueba
44  * si la expresión dada está correctamente balanceada o no, usando una
45  * pila de caracteres delimitadores de apertura
46  */
47 int main() {
48     string expresion;
49     cout << "Introduce una expresión: ";
50     cin >> expresion;
51     cout << "La expresión: " << expresion << endl;
52     if (!balanceada(expresion)){
53         cout << "NO ";
54     }
55     cout << "ESTÁ balanceada." << endl;
56 
57     return 0;
58 }
```