1. Especificación del TAD

```
1  espec PilaGenérica
2  parámetro formal
3      genéro Elemento
4  fpf
5  genéro Pila
6  { Los valores del TAD Pila representan una pila de elementos,
7   que sigue un comportamiento LIFO (Last In, First Out). Los
8   valores de Pila disponen de operaciones que permiten apilar
9   elementos, desapilar el último elemento apilado, consultar
10  el tope de la pila y verificar si la pila está vacía. }
11 operaciones
12     inicializar: -> Pila { operación generadora }
13     { Devuelve una pila vacía }
14
15     apilar: Pila p, Elemento e -> Pila { generadora }
16     { Devuelve la pila resultante de añadir e a p }
17
18     desapilar: Pila p -> Pila { modificadora }
19     { Si p es no vacía, devuelve la pila resultante de eliminar
20     de p el último elemento que fue apilado. En caso contrario,
21     devuelve una pila igual a p }
22
23     cima: Pila p -> Elemento { observadora }
24     { Devuelve el último elemento apilado en la pila p.
25     Parcial: la operación no está definida si estáVacía?(p) }
26
27     estáVacía?: Pila p -> booleano { observadora }
28     { Devuelve verdad si la pila p está vacía (es decir, no
29     contiene ningún elemento), falso en caso contrario }
30
31     { operaciones para el iterador }
32
33     iniciarIterador: Pila p -> Pila
34     { Devuelve una pila igual a la pila p, pero habiendo preparado
35     el iterador para que el siguiente elemento a visitar sea la
36     cima de la pila p, si existe (situación de no haber visitado
37     ningún elemento) }
38
39     existeSiguiente?: Pila p -> booleano
40     { Devuelve falso si ya se han visitado todos los elementos
41     contenidos en p. En caso contrario, devuelve verdad }
42
43     parcial siguiente: Pila p -> Elemento e
44     { Devuelve el siguiente elemento a visitar de la pila p.
45     Parcial: no está definida si not existeSiguiente?(p) }
46
```



```
47      parcial avanza: Pila p -> Pila
48      { Devuelve la pila resultante tras avanzar el iterador
49      de la pila p al siguiente elemento a visitar.
50      Parcial: no está definida si not existeSiguiente?(p) }
51
52  fespec
```

2. Implementación del TAD (pseudocódigo)

```
1  módulo genérico pilasGenéricas
2  parámetro
3      tipo elemento
4  exporta
5      constante
6          TAM_MAX = 1000
7      tipo pila
8      { Los valores del TAD pila representan secuencias de elementos
9      con acceso LIFO (Last In, First Out), esto es, el último
10     elemento añadido será el primero en ser borrado.
11     Implementación limitada a pilas con un tamaño máximo de
12     TAM_MAX elementos }
13
14     procedimiento inicializar(sal p: pila)
15     { Crea una pila vacía (es decir, que no contiene ningún
16     elemento) }
17
18     procedimiento apilar(e/s p: pila; ent e: elemento;
19                              sal error: booleano)
20     { Si p no está llena, devuelve en el mismo parámetro p la pila
21     resultante de añadir e a p y error = falso. En caso contrario,
22     la pila p no se modifica y error = verdad }
23
24     procedimiento desapilar(e/s p: pila; sal e: elemento;
25                                 sal error: booleano)
26     { Si p es no vacía, devuelve en p la pila resultante de
27     eliminar de p el último elemento que fue apilado, dejándolo en
28     e, y error = falso. Si p es vacía, la deja igual, e queda
29     indefinido y error = verdad }
30
31     procedimiento cima(ent p: pila; sal e: elemento;
32                           sal error: booleano)
33     {Si p es no vacía, devuelve en e el último elemento apilado
34     en p y error = falso. Si p es vacía, devuelve error = verdad
35     y e queda indefinido }
36
37     función estáVacía?(p: pila) devuelve booleano
38     { Devuelve verdad si y sólo si p no tiene ningún elemento }
39
40     { operaciones para el iterador }
41
42     procedimiento iniciarIterador(e/s p: pila)
43     { Prepara el iterador para que el siguiente elemento a visitar
44     sea la cima de la pila p, si existe (situación de no haber
45     visitado ningún elemento) }
46
```

```
47      función existeSiguiente?(p: pila) devuelve booleano
48      { Devuelve falso si ya se han visitado todos los elementos
49      contenidos en p, verdad en caso contrario }
50
51      procedimiento siguiente(e/s p: pila; sal e: elemento;
52                      sal error: booleano)
53      { Implementa las ops. siguiente y avanza de la especificación.
54      Si existeSiguiente?(p), error = falso y e toma el valor del
55      siguiente elemento de la pila, y se avanza el iterador al
56      siguiente elemento de la pila. En caso contrario,
57      error = verdad, e queda indefinido y p queda como estaba }
58
59  implementación
60      tipos
61          índice = 0..TAM_MAX  {indica la posición en la pila}
62          pila   = registro
63                       datos: vector[1..TAM_MAX] de elemento;
64                       tope: índice;
65                       iter: índice; { iterador }
66                   freg
67
68      procedimiento inicializar(sal p: pila)
69      { Crea una pila vacía (es decir, que no contiene ningún
70      elemento) }
71      principio
72          p.tope = 0;
73      fin
74
75      función estáLlena?(p: pila) devuelve booleano
76      { Devuelve verdad si la pila p está llena, falso en caso
77      contrario }
78      principio
79          devuelve (p.tope = TAM_MAX)
80      fin
81
82      procedimiento apilar(e/s p: pila; ent e: elemento;
83                           sal error: booleano)
84      { Si p no está llena, devuelve en el mismo parámetro p la pila
85      resultante de añadir e a p y error = falso. En caso contrario,
86      la pila p no se modifica y error = verdad }
87      principio
88          error := verdad;
89          si not estáLlena?(p) entonces
90              p.tope := p.tope + 1;
91              p.datos[p.tope] := e;
92              error := falso;
93          fsi
94      fin
95
```

```
96      procedimiento desapilar(e/s p: pila; sal e: elemento;
97                                  sal error: booleano)
98      { Si p es no vacía, devuelve en p la pila resultante de
99      eliminar de p el último elemento que fue apilado, dejándolo en
100     e, y error = falso. Si p es vacía, la deja igual, e queda
101     indefinido y error = verdad }
102     principio
103         error := verdad;
104         si not estáVacía?(p) entonces
105                   e := p.datos[p.tope];
106               p.tope := p.tope - 1;
107                error := falso;
108         fsi
109     fin
110
111     procedimiento cima(ent p: pila; sal e: elemento;
112                           sal error: booleano)
113     {Si p es no vacía, devuelve en e el último elemento apilado
114     en p y error = falso. Si p es vacía, devuelve error = verdad
115     y e queda indefinido }
116     principio
117         error := verdad;
118         si not estáVacía?(p) entonces
119                   c := p.datos[p.tope];
120               error := falso;
121         fsi
122     fin
123
124     función estáVacía?(p: pila) devuelve booleano
125     { Devuelve verdad si y sólo si p no tiene elementos }
126     principio
127           devuelve (p.tope = 0)
128     fin
129
```

```
130     { Operaciones para el iterador }
131
132     procedimiento iniciarIterador(e/s p: pila)
133     { Prepara el iterador para que el siguiente elemento a visitar
134     sea la cima de la pila p, si existe (situación de no haber
135     visitado ningún elemento) }
136     principio
137         p.iter := p.tope;
138     fin
139
140     función existeSiguiente?(p: pila) devuelve booleano
141     { Devuelve falso si ya se han visitado todos los elementos
142     contenidos en p, verdad en caso contrario }
143     principio
144          devuelve p.iter ≠ 0
145     fin
146
147     procedimiento siguiente(e/s p: pila; sal e: elemento;
148                  sal error: booleano)
149     { Implementa las ops. siguiente y avanza de la especificación.
150     Si existeSiguiente?(p), error = falso y e toma el valor del
151     siguiente elemento de la pila, y se avanza el iterador al
152     siguiente elemento de la pila. En caso contrario,
153     error = verdad, e queda indefinido y p queda como estaba }
154     principio
155         error := verdad;
156         si existeSiguiente?(p) entonces
157             e := p.datos[p.iter];
158             p.iter := p.iter - 1;
159             error := falso;
160         fsi
161     fin
162
163 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 pilasGenéricas, cadenas, ficheros

{ concretización a pila de caracteres }
módulo pilaDeCaracteres = pilasGenéricas(carácter);

    { 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: pilaDeCaracteres.pila;
    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. Implementación en C++

4.1. Fichero «pila_generica.hpp»

```
1  #ifndef _PILAGENERICA_HPP
2  #define _PILAGENERICA_HPP
3
4  // Constantes y tipos previos
5  const unsigned TAM_MAX = 1000;
6
7  // Interfaz del TAD Pila. Pre-declaraciones:
8
9  /* Los valores del TAD Pila representan una pila de elementos,
10  * que sigue un comportamiento LIFO (Last In, First Out). Los
11  * valores de Pila disponen de operaciones que permiten apilar
12  * elementos, desapilar el último elemento, consultar el tope
13  * de la pila y verificar si la pila está vacía.
14  *
15  * El tipo Elemento requerirá tener el operador habitual de
16  * copia =, que se utilizará en las operaciones de apilar,
17  * desapilar y cima.
18  *     Elemento& operator=(const Elemento& origen)
19  */
20 template<typename Elemento> struct Pila;
21
22 /* Crea una pila vacía (es decir, no contiene ningún elemento) */
23 template<typename Elemento> void inicializar(Pila<Elemento>& p);
24
25 /* Si p no está llena, devuelve en el mismo parámetro p la pila
26  * resultante de añadir e a p y error = falso. En caso contrario,
27  * la pila p no se modifica y error = verdad */
28 template<typename Elemento> void apilar(Pila<Elemento>& p,
29                                         Elemento e, bool& error);
30
31 /* Si p no es vacía, devuelve la pila resultante de eliminar
32  * el último elemento que fue apilado, dejándolo en c, y error = falso.
33  * Si la pila es vacía, error = verdad */
34 template<typename Elemento> void desapilar(Pila<Elemento>& p,
35                                         Elemento& e, bool& error);
36
37 /* Si p no es vacía, devuelve en e el último elemento apilado en p y
38  * error = falso. En caso contrario, error = verdad y e queda indefinido */
39 template<typename Elemento> void cima(const Pila<Elemento>& p,
40                                         Elemento& e, bool& error);
41
42 /* Devuelve verdad si y sólo si p no tiene ningún elemento */
43 template<typename Elemento> bool estaVacia(const Pila<Elemento>& p);
44
45 /* Devuelve verdad si y sólo si p ha alcanzado su máximo de capacidad */
46 template<typename Elemento> bool estaLlena(const Pila<Elemento>& p);
47 // Al hacer la pila genérica, el uso de template nos obliga a declarar
48 // esta función como pública :(
49
50 /* Operaciones del iterador */
51
52 /* Prepara el iterador para que el siguiente elemento a visitar
53  * sea la cima de la pila p, si existe (situación de no haber
54  * visitado ningún elemento) */
55 template<typename Elemento> void iniciarIterador(Pila<Elemento>& p);
56
```

```
57  /* Devuelve falso si ya se han visitado todos los elementos
58   * contenidos en p, verdad en caso contrario */
59  template<typename Elemento> bool existeSiguiente(const Pila<Elemento>& p);
60
61  /* Implementa las ops. siguiente y avanza de la especificación.
62   * Si existeSiguiente(p), error = falso y e toma el valor del
63   * siguiente elemento de la pila, y se avanza el iterador al
64   * siguiente elemento de la pila. En caso contrario,
65   * error = verdad, e queda indefinido y p queda como estaba */
66  template<typename Elemento> void siguiente(Pila<Elemento>& p,
67                                             Elemento& e, bool& error);
68
69  // Declaración
70
71  template<typename Elemento> struct Pila {
72      friend void inicializar<Elemento>(Pila<Elemento>& p);
73      friend void apilar<Elemento>(Pila<Elemento>& p, char c, bool& error);
74      friend void desapilar<Elemento>(Pila<Elemento>& p, char& c, bool& error);
75      friend void cima<Elemento>(const Pila<Elemento>& p, char& c, bool& error);
76      friend bool estaVacia<Elemento>(const Pila<Elemento>& p);
77      friend bool estaLlena<Elemento>(const Pila<Elemento>& p);
78      friend void iniciarIterador<Elemento>(Pila<Elemento>& p);
79      friend bool existeSiguiente<Elemento>(const Pila<Elemento>& p);
80      friend void siguiente<Elemento>(Pila<Elemento>& p, Elemento& e, bool& error);
81
82      private: // Representación interna de los valores del TAD
83          Elemento datos[TAM_MAX];
84          int tope;
85          int iter; // iterador
86  };
87
88  // Implementación de las operaciones: al ser un TAD genérico,
89  // el código ha de estar en el archivo HPP (una guarrada, pero OBLIGATORIO)
90
91  /* Crea una pila vacía (es decir, no contiene ningún elemento) */
92  template<typename Elemento> void inicializar(Pila<Elemento>& p) {
93      p.tope = -1;
94      // necesidades de C: los vectores se indexan empezando en 0
95  }
96
97  /* Devuelve verdad si y sólo si p ha alcanzado su máximo de capacidad */
98  template<typename Elemento> bool estaLlena(const Pila<Elemento>& p) {
99    return p.tope == (TAM_MAX - 1);
100 }
101
102 /* Si p no está llena, devuelve en el mismo parámetro p la pila
103  * resultante de añadir e a p y error = falso. En caso contrario,
104  * la pila p no se modifica y error = verdad */
105 template<typename Elemento> void apilar(Pila<Elemento>& p,
106                                         Elemento e, bool& error) {
107     error = true;
108     if (!estaLlena(p)) {
109         p.tope = p.tope + 1;
110         p.datos[p.tope] = e;
111         error = false;
112     }
113 }
114
```

```
115 /* Si p no es vacía, devuelve la pila resultante de eliminar
116  * el último elemento que fue apilado, dejándolo en c, y error = falso.
117  * Si la pila es vacía, error = verdad */
118 template<typename Elemento> void desapilar(Pila<Elemento>& p,
119                                            Elemento& e, bool& error) {
120     error = true;
121     if (!estaVacia(p)) {
122         e = p.datos[p.tope];
123         p.tope = p.tope - 1;
124         error = false;
125     }
126 }
127
128 /* Si p no es vacía, devuelve en e el último elemento apilado en p y
129  * error = falso. En caso contrario, error = verdad y e queda indefinido */
130 template<typename Elemento> void cima(const Pila<Elemento>& p,
131                                       Elemento& e, bool& error) {
132     error = true;
133     if (!estaVacia(p)) {
134         e = p.datos[p.tope];
135         error = false;
136     }
137 }
138
139 /* Devuelve verdad si y sólo si p no tiene ningún elemento */
140 template<typename Elemento> bool estaVacia(const Pila<Elemento>& p) {
141   return (p.tope == -1);
142 }
143
144 /* Operaciones del iterador */
145
146 /* Prepara el iterador para que el siguiente elemento a visitar
147  * sea la cima de la pila p, si existe (situación de no haber
148  * visitado ningún elemento) */
149 template<typename Elemento> void iniciarIterador(Pila<Elemento>& p) {
150     p.iter = p.tope;
151 }
152
153 /* Devuelve falso si ya se han visitado todos los elementos
154  * contenidos en p, verdad en caso contrario */
155 template<typename Elemento> bool existeSiguiente(const Pila<Elemento>& p) {
156     return p.iter != -1;
157 }
158
159 /* Implementa las ops. siguiente y avanza de la especificación.
160  * Si existeSiguiente(p), error = falso y e toma el valor del
161  * siguiente elemento de la pila, y se avanza el iterador al
162  * siguiente elemento de la pila. En caso contrario,
163  * error = verdad, e queda indefinido y p queda como estaba */
164 template<typename Elemento> void siguiente(Pila<Elemento>& p,
165                                            Elemento& e, bool& error) {
166     error = true;
167     if(existeSiguiente(p)) {
168         e = p.datos[p.iter];
169         p.iter--;
170         error = false;
171     }
172 }
173
174 #endif
```

4.2. Fichero «main.cpp» (uso del TAD)

```
1  #include <iostream>
2  #include <string>
3  #include "pila_generica.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 una Pila (TAD genérico)
21  * concretizada con elementos de tipo char.
22  * Ojo: se asume que la cadena vacía es una expresión balanceada */
23 bool balanceada(string s) {
24     Pila<char> p;
25     bool error = false,
26          estaBalanceada = true;
27     unsigned len = s.length();
28
29     inicializar(p);
30
31     for (unsigned i = 0; i < len && !error && estaBalanceada; i++){
32         char c = s[i], c2;
33         if (c == '(' || c == '[' || c == '{') {
34             apilar(p, c, error);
35         } else if (c == ')' || c == ']' || c == '}') {
36             desapilar(p, c2, error);
37             estaBalanceada = (cierre(c2) == c);
38         }
39     }
40     return estaBalanceada && estaVacia(p);
41 }
42
43 /*
44  * Programa principal que solicita una expresión al usuario y comprueba
45  * si la expresión dada está correctamente balanceada o no, usando una
46  * pila de caracteres delimitadores de apertura
47  */
48 int main() {
49     string expresion;
50     cout << "Introduce una expresión: ";
51     cin >> expresion;
52     cout << "La expresión: " << expresion << endl;
53     if (!balanceada(expresion)){
54         cout << "NO ";
55     }
56     cout << "ESTÁ balanceada." << endl;
57
58     return 0;
59 }
```