1. Una pila (stack en inglés) es una estructura de datos que permite almacenar y recuperar datos, siendo la semántica de acceso a sus elementos del tipo **LIFO** (del inglés **L**ast **I**n, **F**irst **O**ut, «último en entrar, primero en salir»). Esta estructura es de uso frecuente en el área de informática debido a su simplicidad y capacidad de dar respuesta a numerosos problemas.
   Para el manejo de los datos cuenta con dos operaciones básicas: **`push`**, que coloca un nuevo objeto en la pila, y su operación inversa, **`pop`**, que retira el último elemento apilado. Puede existir además una operación para chequear si el stack está o no vacío.
   En cada momento sólo se tiene acceso a la parte superior de la pila, es decir, al último objeto apilado (denominado **TOS**: **T**op **o**f **S**tack en inglés). La operación retirar (**`pop`**) permite obtener este elemento, que es retirado de la pila permitiendo el acceso al anterior (apilado con anterioridad), que pasa a ser el último, el nuevo TOS.
   Se solicita implementar el UDT **`Stack`** para almacenar caracteres con su interface básica:

```
class Stack {
    public:
        void push(char c);  // coloca un nuevo carácter en la pila
        char pop();         // retira el elemento al tope del stack
        bool isEmpty();     // retorna true/false indicando si el stack está vacío
    private:
        // ...
        // ...
};
```

2. Se desea implementar una agenda telefónica en donde cada entrada de la agenda mantenga a una instancia del UDT **`Persona`**, que cuenta con los datos típicos que describen a una persona, entre ellos: su documento, nombre, domicilio y número de teléfono. Se desea dar a la agenda capacidad de búsqueda a partir del DNI o del nombre de la persona a buscar. En cualquiera de los casos, si la agenda encuentra la persona buscada, retornará un puntero a esa persona, o **`nullptr`** para indicar que la persona no está registrada en la agenda.

```
struct Persona {
    // ...
    // ...
};

class Agenda {
    public:
        void agregaPersona(const Persona &p);
        const Persona *buscaPorDNI(unsigned int dni);
        const Persona *buscaPorNombre(const string &nombre);

    private:
        vector<Persona> personas;
};
```

3. Que modificaciones haría en el ejercicio anterior si desea hacer mucho más eficientes las búsquedas, tanto por nombre como por dni? Impleméntelas.

4. Una forma de analizar la "correctitud sintáctica" de una expresión algebraica en lo que respecta a apertura y cierre de signos de agrupación (paréntesis, corchetes y llaves) es utilizar un **`Stack`** y:
   a) Recorrer la expresión (string) caracter a caracter.
      i. Cuando se encuentra una apertura de agrupación, se agrega al **`Stack`** el signo de apertura.
      ii. Cuando se encuentra un cierre de agrupación, el **`Stack`** no puede estar vacío y el signo que se remueve del mismo se debe corresponder con el signo de cierre que se está procesando.
   b) Cuando se completa el recorrido el **`Stack`** debe quedar vacío.

   Con esta información, implementar la función:

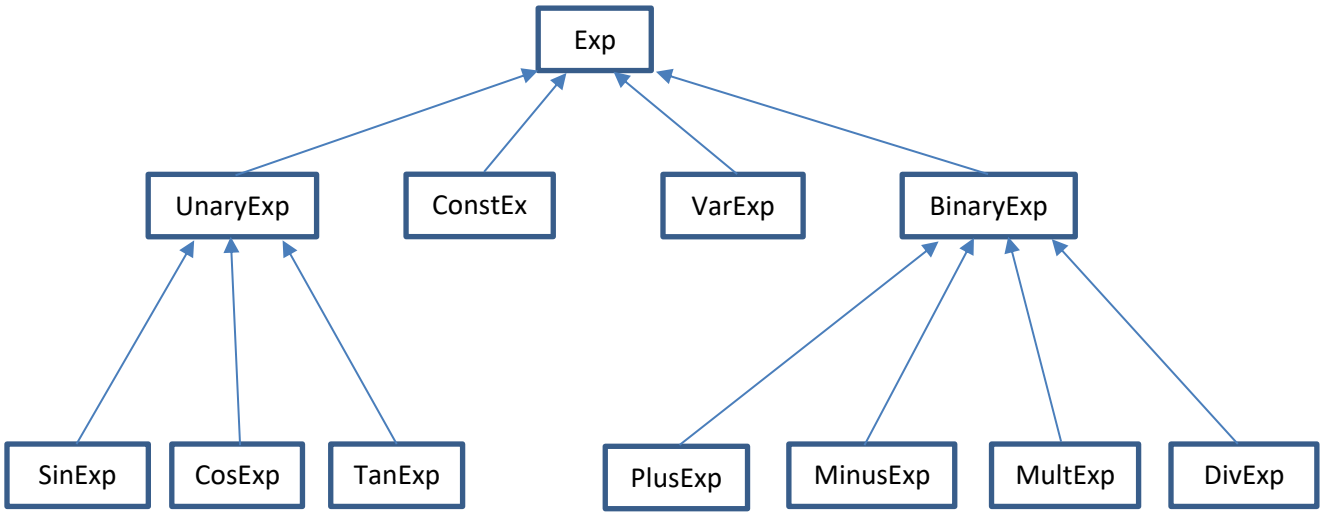
   ```
   bool CheckExpresion(const string &expr);
   ```

   Probar la función con expresiones como:

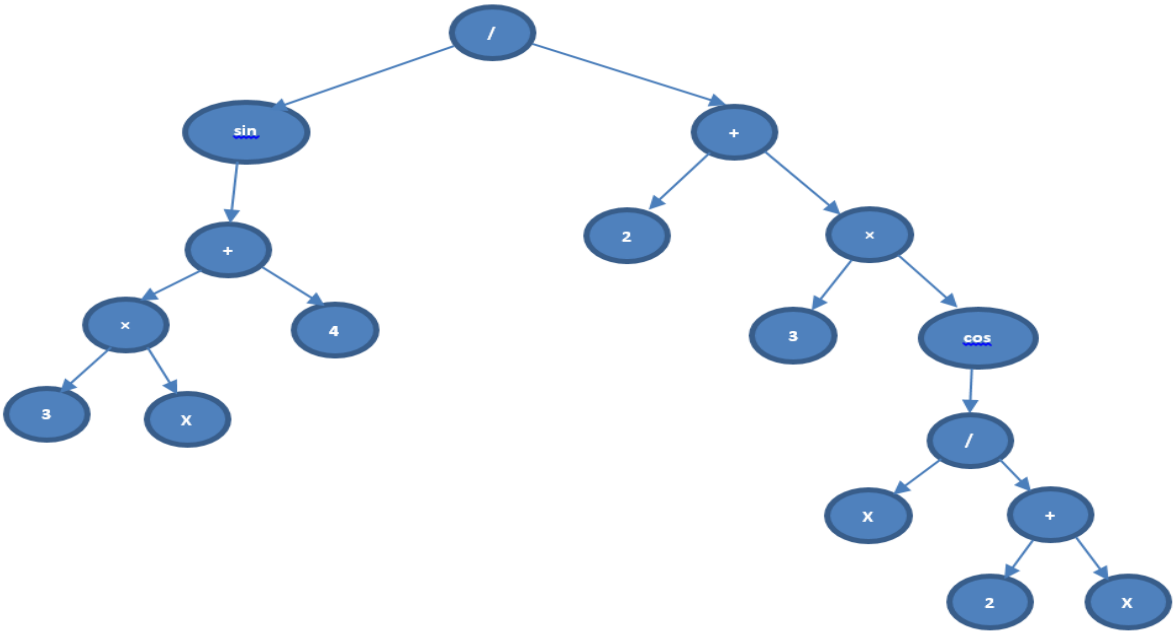
   A+Y*{12+z*[sin(34+PI*(2+Y)) + 3] + Q} + (4*Y)

   U otras que tengan errores.

5. En el esqueleto de código presente en **`expression.cpp`** se implementa una jerarquía como la que se muestra en el diagrama siguiente para representar expresiones algebraicas:



Implemente los métodos faltantes y todo otro método que crea necesario. Pruebe el código con el **`main`** dado. En ese **`main`**, se está creando la expresión: $\frac{\sin(3x+4)}{2+3\cos\left(\frac{x}{2+x}\right)}$ que puede verse gráficamente como:



Investigue una forma alternativa para poder escribir el main con una notación más humana.