INFORME DE LABORATORIO

(formato estudiante)

INFORMACIÓN BÁSICA					
ASIGNATURA:					
TÍTULO DE LA PRÁCTICA:					
NÚMERO DE PRÁCTICA:		**AÑO LECTIVO:**		**NRO. SEMESTRE:**	
FECHA DE PRESENTACIÓN		**HORA DE PRESENTACIÓN**			
INTEGRANTE (s):				**NOTA:**	
DOCENTE(s):					

SOLUCIÓN Y RESULTADOS

I. SOLUCIÓN DE EJERCICIOS/PROBLEMAS

DRIVE: https://drive.google.com/drive/folders/1rpabolY3Wu7S9v4qh8OfYG0D7dUF24it?usp=drive_link

1 Objetivos

- *El alumno deberá de manipular diferentes tipos de estructuras utilizando punteros (pointers)*

2 Temas a Tratar

- *Manejo de punteros en c++*
- *Administración de memoria en c++*
- *Arbol binario de expresión*

3 Actividades

3.1 Partiendo de la implementacion de la Calculadora utilizando Clases. El alumno deberá de añadir la funcionalidad de multiplicacion en el programa elaborado.

3.2 El alumno deberáde implementar la clase multiplicacion.

La clase Multiplicación implementa el patrón de diseño **Composite Pattern** junto con **polimorfismo**. Al heredar de la clase abstracta Operacion, garantizamos que todos los tipos de operación tengan una interfaz uniforme. Los punteros operando1 y operando2 permiten que cada operando sea cualquier tipo de operación (números, sumas, restas, etc.), creando una estructura recursiva.

C/C++

```
// Clase para multiplicación
class Multiplicacion : public Operacion {
private:
    Operacion* operando1;
    Operacion* operando2;

public:
    Multiplicacion(Operacion* op1, Operacion* op2) : operando1(op1), operando2(op2) {}

    double calcular() override {
        return operando1->calcular() * operando2->calcular();
    }

    ~Multiplicacion() {
        delete operando1;
        delete operando2;
    }
};
```

3.3 Dentro del programa elaborado, se deberáde actualizar los objetos de clases por punteros.

El cambio de objetos por valor a punteros implementa gestión manual de memoria. Esto es crucial cuando trabajamos con estructuras de datos complejas como árboles, donde el tamaño y la estructura pueden variar dinámicamente durante la ejecución.

C/C++

```
class Calculadora {
private:
    ArbolExpresion* arbol;

public:
    Calculadora() {
        arbol = new ArbolExpresion();
    }

    ~Calculadora() {
        delete arbol;
    }
```

```
};
```

*3.4 Respecto a la cadena de texto de entrada, esta contendrá el símbolo de multiplicacion denotado por el simbolo asterisco (*), la cual estará presente en la operacion a realizar ("33+1","2*5","12+4*12","34*3+1*90")*

C/C++

```
int precedencia(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;  // Mayor precedencia
    return 0;
}

switch (nodo->operador) {
    case '*': return izq * der;
    // ...otros casos
}
```

La implementación del operador multiplicación requiere considerar las **reglas de precedencia matemática**. Al asignar precedencia 2 a * y /, y precedencia 1 a + y -, garantizamos que las multiplicaciones se evalúen antes que las sumas, respetando el orden matemático estándar.

3.5 El alumno deberá de implementar un árbol binario de expresion (Binary expression tree Fig. 1) utilizando clases y punteros, para poder operar las operaciones de suma y multiplicacion en el orden correcto.

C/C++

```
struct NodoArbol {
    char operador;
    double numero;
    bool esOperador;
    NodoArbol* izquierdo;
    NodoArbol* derecho;

    ~NodoArbol() {
        delete izquierdo;
        delete derecho;
```

```
    }
};
```

El árbol binario implementa una estructura de datos recursiva donde cada nodo puede ser either un operador (nodo interno) o un número (nodo hoja). Esta dualidad se maneja con el flag esOperador y permite representar expresiones matemáticas de cualquier complejidad.

3.6 Aplicar lo aprendido en las clases de teoria.

```
C/C++
NodoArbol* construirDesdePostfija(string postfija) {
    stack<NodoArbol*> pila;
    // Procesamiento token por token
    while (iss >> token) {
        if (token == "+" || token == "-" || token == "*" || token == "/") {
            NodoArbol* nodo = new NodoArbol(token[0]);
            nodo->derecho = pila.top(); pila.pop();
            nodo->izquierdo = pila.top(); pila.pop();
            pila.push(nodo);
        }
    }
}
```

La notación postfija elimina la ambigüedad de precedencia. En postfija, 2 3 4 * + se lee como "toma 3 y 4, multiplícalos, luego suma 2", lo que naturalmente construye el árbol correcto sin necesidad de considerar precedencia durante la construcción.

Figure 1: Binary expression tree

4 Actividades Extra

4.1 Implementar operación de resta y división

- *Crear las clases Resta y División.*
- *Modificar el árbol binario de expresiones para soportar también estos operadores.*
- *Considerar validación para división entre cero.*

```
C/C++
class Division : public Operacion {
    double calcular() override {
        double divisor = operando2->calcular();
        if (divisor == 0) {
            throw runtime_error("Error: División entre cero");
        }
        return operando1->calcular() / divisor;
    }
};
```

La validación de división por cero implementa manejo de excepciones en lugar de retornar valores especiales (como NaN o infinito). Esto sigue el principio de "fail fast": es mejor detectar y reportar errores inmediatamente que continuar con datos corruptos.
Las excepciones permiten separar la lógica de negocio del manejo de errores. El código que llama a calcular() puede decidir cómo manejar el error (mostrar mensaje, reintentar, etc.) sin ensuciar la lógica matemática con validaciones.

4.2 Recorrido del árbol binario

• Implementar métodos para recorrer el árbol en:

o Preorden

o Inorden

o Postorden

• Mostrar en consola las operaciones según cada recorrido.

```
C/C++
void inorden(NodoArbol* nodo) {
    if (!nodo) return;
    inorden(nodo->izquierdo);
    // Procesar nodo actual
    inorden(nodo->derecho);
}
```

Los tres recorridos implementan diferentes **estrategias de traversal**:

- **Inorden (LNR)**: Produce la expresión en notación infija original
- **Preorden (NLR)**: Produce notación prefija (operador antes que operandos)
- **Postorden (LRN)**: Produce notación postfija (operandos antes que operador)

Cada recorrido tiene aplicaciones específicas:

- **Inorden**: Para reconstruir la expresión original
- **Preorden**: Para evaluar expresiones de forma top-down
- **Postorden**: Para evaluación eficiente (como lo hacen las calculadoras)

4.3 Evaluación paso a paso

• Modificar la clase del árbol para que, además de devolver el resultado final, muestre cómo se resuelve la expresión paso a paso.

• Ejemplo:

*Entrada: 12+4*3*

*Paso 1: 4*3 = 12*

Paso 2: 12+12 = 24.

```C/C++
double evaluarPasoAPaso(NodoArbol* nodo, int& paso) {
    if (!nodo->esOperador) return nodo->numero;

    double izq = evaluarPasoAPaso(nodo->izquierdo, paso);
    double der = evaluarPasoAPaso(nodo->derecho, paso);
    double resultado = // calcular según operador

    cout << "Paso " << ++paso << ": " << izq << " " << nodo->operador
         << " " << der << " = " << resultado << endl;
    return resultado;
}
```

Esta función implementa evaluación con trazabilidad. El parámetro paso se pasa por referencia (int&) para mantener un contador global que persiste entre llamadas recursivas. Esto permite mostrar el orden exacto de evaluación.
La evaluación paso a paso tiene valor pedagógico: permite entender cómo la computadora resuelve expresiones complejas. También es útil para debugging: si el resultado final es incorrecto, podemos ver exactamente dónde ocurrió el error.

4.4 Manejo dinámico de memoria

• Usar new y delete para crear y destruir dinámicamente los nodos del árbol.

• Demostrar con un método que libere correctamente toda la memoria (función destruirArbol).

```
C/C++
void destruirArbol(NodoArbol* nodo) {
    if (nodo) {
        destruirArbol(nodo->izquierdo);
        destruirArbol(nodo->derecho);
        delete nodo;
    }
}

~ArbolExpresion() {
    destruirArbol(raiz);
}
```

La función destruirArbol implementa liberación recursiva de memoria. Sigue el patrón post-order: primero libera los hijos, luego el nodo padre. Esto evita acceder a memoria ya liberada.

¿Por qué es crítico el manejo de memoria? En C++, toda memoria asignada con new debe liberarse con delete. Sin liberación apropiada, el programa sufre memory leaks. En aplicaciones que procesan muchas expresiones, estos leaks pueden consumir toda la memoria disponible. El destructor automático garantiza que la memoria se libere incluso si ocurre una excepción.

4.5 Historial de expresiones evaluadas

• Almacenar en un arreglo dinámico (puntero a puntero de char o string) las últimas operaciones ingresadas y sus resultados.

• Mostrar el historial al finalizar el programa.

```
C/C++
vector<string> historial;

void agregarAlHistorial(string expresion, double resultado) {
    historial.push_back(expresion + " = " + to_string(resultado));
}
```

El historial usa std::vector que maneja automáticamente el redimensionamiento. Al concatenar expresión y resultado en un solo string, simplificamos el almacenamiento y la visualización.

Los vectores ofrecen gestión automática de memoria y tamaño dinámico. No necesitamos saber de antemano cuántas expresiones evaluará el usuario. Además, vector proporciona iteradores que simplifican el recorrido para mostrar el historial.

4.6 Implementar paréntesis en expresiones

*• Extender el programa para aceptar expresiones con paréntesis ((12+3)*4).*

• Modificar el parser que construye el árbol binario de expresiones para que respete la jerarquía de operaciones.

```C/C++
else if (c == '(') {
    operadores.push(c);
}
else if (c == ')') {
    while (!operadores.empty() && operadores.top() != '(') {
        resultado += operadores.top();
        resultado += " ";
        operadores.pop();
    }
    operadores.pop(); // Quitar el '('
}
```

Los paréntesis implementan precedencia forzada. El algoritmo de Shunting Yard trata (como un operador de precedencia infinita que nunca se pop hasta encontrar su)correspondiente. Esto permite expresiones como (2+3)*4 donde la suma debe evaluarse antes que la multiplicación, contrario a la precedencia normal.

Los paréntesis pueden anidarse arbitrariamente: ((2+3)*(4-1))+5. Cada nivel de anidamiento requiere mantener el estado de qué operadores están "suspendidos" esperando el cierre del paréntesis. La pila maneja este anidamiento naturalmente.

4.7 Conversión de expresiones

• Implementar métodos para convertir la expresión:

*o De infija (ej. 3+4*2)*

*o A postfija (RPN) (ej. 3 4 2 * +)*

*o A prefija (ej. + 3 * 4 2).*

```
C/C++
string convertirAPrefija(string expresion) {
    // Invertir expresión y cambiar paréntesis
    string invertida = "";
    for (int i = expresion.length() - 1; i >= 0; i--) {
        if (expresion[i] == '(') invertida += ')';
        else if (expresion[i] == ')') invertida += '(';
        else invertida += expresion[i];
    }

    // Obtener postfija de la invertida
    string postfija = infijaAPostfija(invertida);

    // Invertir resultado
    // ...
}
```

La conversión a prefija usa un truco algorítmico elegante:

1. Invertir la expresión infija cambiando () por)()
2. Aplicar el algoritmo estándar infija→postfija
3. Invertir el resultado

Esto funciona porque la notación prefija es esencialmente postfija "al revés".

Diferentes notaciones tienen diferentes aplicaciones:

- Infija: Natural para humanos (2+3*4)
- Postfija: Eficiente para computadoras (no requiere paréntesis ni precedencia)
- Prefija: Útil en programación funcional y algunos compiladores

El programa demuestra cómo convertir entre estas notaciones, mostrando la flexibilidad de la representación interna en árbol.

4.8 Uso de punteros inteligentes (opcional, avanzado)

• Reemplazar el uso de punteros crudos (*) por punteros inteligentes de C++11

(std::unique_ptr, std::shared_ptr).

• Explicar las ventajas frente al manejo manual de memoria.

```C/C++
class Suma : public Operacion {
private:
    std::unique_ptr<Operacion> operando1;
    std::unique_ptr<Operacion> operando2;

public:
    Suma(std::unique_ptr<Operacion> op1, std::unique_ptr<Operacion> op2)
        : operando1(std::move(op1)), operando2(std::move(op2)) {}

    double calcular() override {
        return operando1->calcular() + operando2->calcular();
    }

    // ¡NO necesita destructor! Se libera automáticamente
};
```

Ventajas de los punteros inteligentes:

1. ***Gestión automática**: No más new/delete manuales*
2. ***Seguridad**: Imposible tener memory leaks o double-delete*
3. ***Semántica clara**: unique_ptr = propiedad exclusiva, shared_ptr = propiedad compartida*
4. ***Exception safety**: Se liberan automáticamente aunque ocurran excepciones*

II. SOLUCIÓN DEL CUESTIONARIO

III. CONCLUSIONES

RETROALIMENTACIÓN GENERAL

REFERENCIAS Y BIBLIOGRAFÍA