Reverse Polish Notation

Requisitos del programa:

- **Nombre del programa:** Debe llamarse **RPN**.

Entrada:

El programa recibirá como argumento una expresión matemática escrita en notación polaca inversa (Reverse Polish Notation, o RPN).

Nota: La notación polaca inversa es aquella en la que primero se colocan los operandos y, a continuación, el operador que se aplicará.

- **Restricciones sobre los números:**

  Los números que se usan en la operación (y que se pasan como argumento) serán siempre menores que 10. (Esta restricción se aplica únicamente a los operandos, no a los resultados intermedios o finales.)

- **Procesamiento:**

  El programa debe analizar la expresión recibida, evaluarla correctamente y mostrar el resultado en la salida estándar.

- **Manejo de errores:**

  Si se produce algún error durante la ejecución (por ejemplo, falta de operandos o un formato no esperado), el programa debe mostrar un mensaje de error en la salida de errores estándar.

- **Operadores admitidos:**

  El programa debe poder manejar los siguientes tokens: +, -, * y /.

- **Uso de contenedores:**

  Es obligatorio utilizar al menos un contenedor de la STL para validar este ejercicio.
  IMPORTANTE: Los contenedores que se hayan usado en ejercicios anteriores están prohibidos en este ejercicio y no podrán utilizarse para el resto del módulo.

Ejemplos de uso:

- Ejecutando:

  $> ./RPN "8 9 * 9 - 9 - 9 - 4 - 1 +"

El programa debe mostrar: 42

- Ejecutando:

  $> ./RPN "7 7 * 7 -"

  El resultado debe ser: 42

- Ejecutando:

  $> ./RPN "1 2 * 2 / 2 * 2 4 - +"

  El resultado debe ser: 0

- Ejecutando:

  $> ./RPN "(1 + 1)"

  Se debe mostrar un mensaje de error.

Propósito de Aprendizaje

El ejercicio tiene como objetivos principales:

1. **Evaluación de Expresiones Postfijas:**

   Aprender a procesar y evaluar expresiones en notación polaca inversa, lo que implica comprender cómo se almacenan y se manipulan los operandos y operadores para obtener un resultado correcto.

2. **Uso de Contenedores de la STL:**

   Aunque la expresión podría evaluarse sin contenedores, en este módulo se enfatiza la correcta utilización de los contenedores estándar (como std::vector, std::list o incluso std::deque) para almacenar datos. Dado que los contenedores usados anteriormente están prohibidos, se debe optar por otro contenedor que cumpla la función de simular una pila.

3. **Manejo de Errores y Validación de Entrada:**

   Es fundamental que el programa detecte y gestione adecuadamente errores en la expresión (por ejemplo, operadores sin suficientes operandos o formato incorrecto).

4. **Aplicación de C++98:**

   La solución debe implementarse siguiendo las restricciones y sintaxis del estándar C++98, lo cual implica, entre otros aspectos, declarar y definir correctamente clases, usar iteradores y contenedores de la STL sin las mejoras de C++11 o superiores.

Cómo Abordar el Ejercicio en C++98

1. Análisis y Parsing de la Entrada

- **Recoger el argumento:**

  El programa debe comprobar que se ha pasado exactamente un argumento (la expresión RPN). Si no, se mostrará un mensaje de error.

- **División de la expresión:**

  La expresión se suministra en una única cadena de caracteres. Se debe dividir esta cadena en tokens (usando, por ejemplo, std::istringstream y el método operator>>) para separar los operandos y operadores.

2. Uso de un Contenedor para Simular una Pila

- **Selección del contenedor:**

  Aunque la opción natural para evaluar expresiones es usar std::stack, si dicho contenedor se utilizó en ejercicios anteriores, se deberá optar por otro contenedor de la STL (por ejemplo, un std::vector o std::deque) para almacenar los elementos de la “pila”.

Término en inglés: **stack** (pila).

- **Implementación de la lógica de pila:**
  - Cuando se encuentra un número (operando), se "apila" (push) en el contenedor.
  - Cuando se encuentra un operador, se deben extraer (pop) los dos últimos elementos del contenedor, aplicar el operador y almacenar (push) el resultado de nuevo en el contenedor.
- **Validación:**

  Se debe comprobar que existan suficientes operandos antes de intentar aplicar un operador. En caso de error, se debe emitir el mensaje adecuado en std::cerr.

3. Evaluación de la Expresión

- **Operaciones:**

  Se deben implementar las operaciones aritméticas básicas (+, -, *, /). Para cada operador, se extraen dos operandos y se aplica la operación en el orden correcto (recordando que la división y la resta no son conmutativas).

- **Conversión de tokens:**

Los tokens numéricos se deben convertir de std::string a tipo numérico (por ejemplo, int) utilizando std::istringstream o funciones similares.

4. Manejo de Errores

- **Errores de formato:**

  Si el token no representa ni un número ni uno de los operadores permitidos, se mostrará un mensaje de error.

- **Errores de operandos insuficientes:**

  Si al aplicar un operador no hay suficientes operandos en la "pila", se deberá emitir un error en la salida de errores estándar.

- **Restricción de operandos:**

  Recordar que los operandos siempre serán menores que 10. Se debe validar este aspecto al procesar la entrada.

5. Salida del Resultado

- Una vez evaluada la expresión por completo, el contenedor debe contener exactamente un elemento, que es el resultado de la operación. Se muestra este resultado en la salida estándar.

Recursos en Video Recomendados (en Español)

Para complementar los conceptos y técnicas necesarios para abordar este ejercicio, se recomienda revisar los siguientes videos:

1. **POLACA INVERSA - NOTACIÓN INFIJA Y POSTFIJA**
   a. *Descripción:* Video tutorial que explica detalladamente la notación polaca inversa y cómo se relaciona con la notación infija y prefija, ideal para entender la evaluación de expresiones.
   b. Ver video
2. **Curso de C++ Avanzado. STL. Contenedores de la STL**
   a. *Descripción:* Este video profundiza en el uso de los contenedores de la STL, mostrando ejemplos y aplicaciones prácticas, lo que será muy útil para elegir un contenedor adecuado en este ejercicio.
   b. Ver video
3. **Curso de C++ Avanzado. STL. Iteradores de la STL**
   a. *Descripción:* Explicación sobre el uso de iteradores en los contenedores de la STL, esencial para manipular y recorrer los datos almacenados en el contenedor elegido.
   b. Ver video

Resumen

El ejercicio “RPN” consiste en desarrollar un programa en C++98 que reciba como argumento una expresión matemática en notación polaca inversa, la evalúe utilizando un contenedor (diferente del que se usó en ejercicios anteriores) para simular el funcionamiento de una pila, y muestre el resultado en la salida estándar. Además, el programa debe gestionar errores en la entrada y cumplir con las restricciones específicas (por ejemplo, operandos menores que 10 y únicamente los operadores +, -, * y /).

El propósito de aprendizaje es familiarizarse con la evaluación de expresiones en notación postfija, el uso adecuado de contenedores de la STL y el manejo de errores, todo ello aplicado en el contexto de C++98.

El código

El código implementa una clase llamada **RPN** que se encarga de validar y evaluar expresiones en notación polaca inversa (RPN – Reverse Polish Notation). A grandes rasgos, el programa realiza las siguientes tareas:

1. Estructura de la Clase RPN

- **Constructores y Operador de Asignación:**

  La clase dispone de un constructor por defecto, un constructor de copia y un operador de asignación. Esto permite crear instancias de RPN y, aunque en este caso no se almacena ningún dato miembro (la única variable interna es la “pila” que se crea localmente en la función calculator), se sigue la buena práctica de definir estos métodos para futuras extensiones.

- **Métodos Estáticos:**

  - **validExpression(const std::string &expr):**

    Este método recibe una cadena con la expresión RPN y comprueba si es válida.

    - Se usa un std::stringstream para leer token a token la expresión.

    - Cada token se evalúa:

      - Si contiene solo dígitos (se verifica con find_first_not_of("0123456789")), se incrementa el contador de números.

- Si es uno de los operadores permitidos (+, -, *, /), se incrementa el contador de operadores. Además, se verifica que haya al menos dos números acumulados en ese momento (pues para aplicar un operador se necesitan dos operandos) y se decrementa el contador de números (porque al aplicar la operación, dos operandos se reemplazan por uno solo).
- Si se encuentra un token que no es ni un número ni un operador, la función retorna false.

▪ Finalmente, la expresión se considera válida si al terminar se tiene exactamente un número (el resultado final) y se ha utilizado al menos un operador.

- **calculator(const std::string &expr):**

  Este método evalúa la expresión RPN y devuelve el resultado como un valor de tipo long long.

  ▪ Se utiliza un std::stack<int> para almacenar los operandos.
  ▪ Se vuelve a usar un std::stringstream para leer cada token.
  ▪ Si el token es un operador, se verifica que en la pila existan al menos dos elementos; de lo contrario, se lanza una excepción del tipo invalidExpression.
  ▪ Se extraen los dos operandos (recordando que en RPN el primer operando extraído es el segundo en la operación, y el segundo extraído es el primero, lo cual es importante para operaciones no conmutativas como la resta y la división).
  ▪ Se evalúa el operador mediante un switch:
    - Para '+': suma los operandos.
    - Para '-': realiza la resta.
    - Para '*': multiplica.
    - Para '/': primero comprueba si el divisor es cero y, en ese caso, lanza una excepción zeroDivision; si no, divide.
  ▪ Tras calcular la operación, se empuja (push) el resultado nuevamente en la pila.
  ▪ Una vez procesados todos los tokens, si la pila no contiene exactamente un elemento, se lanza la excepción invalidExpression; en caso contrario, se retorna el valor final.

- **Excepciones Personalizadas:**

  La clase define dos clases de excepción anidadas que heredan de std::exception:

  - **invalidExpression:** Se lanza cuando la expresión no es válida (por ejemplo, si no hay suficientes operandos o la pila no termina con un único resultado).

- **zeroDivision:** Se lanza cuando se intenta dividir entre cero. Cada una redefine el método what() para devolver un mensaje de error formateado (usando colores definidos con macros).

2. Función Principal (main.cpp)

- **Validación de Argumentos:**

  La función main verifica que se haya pasado exactamente un argumento (la expresión RPN). Si no es así, se llama a la función evil, que muestra un mensaje de error y finaliza el programa con un código de error.

- **Validación de la Expresión:**

  Se utiliza el método estático RPN::validExpression para comprobar que la cadena de entrada es una expresión RPN válida. Si la validación falla, se emite un error.

- **Presentación y Evaluación:**

  Se muestra en pantalla un "banner" decorativo (usando colores y líneas) y luego, en un bloque try-catch, se llama a RPN::calculator para evaluar la expresión.

  - Si la evaluación tiene éxito, se imprime el resultado.
  - Si se lanza alguna excepción (por expresión inválida o división por cero), se captura y se muestra el mensaje de error asociado.

- **Salida del Programa:**

  Finalmente, el programa finaliza con un código de éxito (EXIT_SUCCESS).

3. Conceptos Clave y Cómo Funciona

- **Notación Polaca Inversa (RPN):**

  En esta notación, los operandos se escriben primero y luego el operador. Por ejemplo, la expresión infija "3 + 4" se escribe en RPN como "3 4 +".

  La evaluación se realiza utilizando una pila: se empujan los números y, cuando se encuentra un operador, se extraen los dos últimos números para aplicar la operación y se empuja el resultado.

- **Uso de std::stack:**

  La clase utiliza el contenedor std::stack de la STL para gestionar la "pila" de operandos, lo cual simplifica la implementación de la lógica LIFO (Last In, First Out).

- **Conversión de Cadenas a Números:**

  La función auxiliar ft_stoi (similar a la función stoi de C++11, pero implementada aquí con std::stringstream) convierte un token numérico (en forma de std::string) a un entero.

- **Manejo de Excepciones:**

  El uso de excepciones permite separar la lógica de evaluación de la gestión de errores. Si se produce alguna anomalía (como dividir entre cero o una expresión mal formada), se lanza una excepción que se captura en main y se muestra un mensaje adecuado.

- **Validación Previa:**

  Antes de evaluar la expresión, se recorre la cadena para contar operandos y operadores. Esto asegura que la expresión tiene la forma correcta (por ejemplo, que al menos haya dos operandos para cada operador y que al final quede un solo resultado).

Conclusión

El código del archivo **RPN.hpp** y su implementación en **RPN.cpp** junto con **main.cpp** conforman un evaluador de expresiones en notación polaca inversa. Se encarga de validar la expresión, convertirla y evaluarla utilizando una pila y controlando errores mediante excepciones personalizadas. Esta implementación es un ejemplo claro de cómo se puede utilizar la STL en C++ (incluyendo el contenedor std::stack y las excepciones) para resolver un problema clásico de evaluación de expresiones aritméticas siguiendo el estándar C++98.