Informe de Laboratorio 03

Tema: C++ Herencia, Polimorfismo, Qt

Nota

Estudiante	Escuela	Asignatura
Reyser Julio Zapata Butron rzapata@unsa.edu.pe	Escuela Profesional de Ingeniería de Sistemas	Tecnología de Objetos Semestre: VI Código: 1703240

Laboratorio	Tema	Duración
03	C++ Herencia, Polimorfismo, Qt	02 horas

Semestre académico	Fecha de inicio	Fecha de entrega
2024 - B	03 octubre 2024	09 octubre 2024

1. Repositorio de Github

- Repositorio de Github donde se encuentra el actual laboratorio
  `https://github.com/ReyserLynnn/tec-obj-lab-c-24b/tree/main/laboratorio03/src`
- Repositorio de Github donde se encuentran los laboratorios del curso
  `https://github.com/ReyserLynnn/tec-obj-lab-c-24b.git`

2. Ejercicios

En los siguientes ejercicios, se presentará código, captura de ejecución y una explicación en general por cada ejercicio.

2.1. Ejercicio 1

- Modificar el ejercicio 01 (rectángulo) para que funcione en la memoria dinámica. Los miembros de la clase deben ser punteros, las instancias en el main deben cargarse en memoria dinámica, el constructor debe crear la instancia en la memoria dinámica. Utilizar punteros inteligentes.

```
#ifndef RECTANGULO_H
#define RECTANGULO_H
#include <memory>
#include <iostream>

```

```
typedef unsigned short int USHORT;
using namespace std;

class Rectangulo
{
private:
  unique_ptr<USHORT> width;
  unique_ptr<USHORT> height;
  unique_ptr<int> temperatura;
  unique_ptr<int> presion;
  unique_ptr<int> humedad;

public:
  Rectangulo();
  Rectangulo(USHORT width, USHORT height);
  Rectangulo(const Rectangulo &R);
  ~Rectangulo() {}

  void Draw();
  void Draw(USHORT width, USHORT height) const;
};

#endif // RECTANGULO_H
```

Listing 1: rectangulo.h

```
#include "rectangulo.h"

Rectangulo::Rectangulo()
    : width(make_unique<USHORT>(1)), height(make_unique<USHORT>(1)),
      temperatura(make_unique<int>(0)), presion(make_unique<int>(0)), humedad(make_unique<int>(0)) {}

Rectangulo::Rectangulo(USHORT w, USHORT h) : width(make_unique<USHORT>(w)), height(make_unique<USHORT>(h
                                             temperatura(make_unique<int>(0)), presion(make_unique<int>(

Rectangulo::Rectangulo(const Rectangulo &R) : width(make_unique<USHORT>(*R.width)), height(make_unique<U
                                              temperatura(make_unique<int>(*R.temperatura)), presion(mak

void Rectangulo::Draw()
{
  // cout << "\n[*} Dibujando un rectangulo de " << width << "x" << height << endl;
  *temperatura = 25;
  *presion = 1013;
  *humedad = 50;

  Draw(*width, *height);
}

void Rectangulo::Draw(USHORT w, USHORT h) const
{
  for (USHORT i = 0; i < h; i++)
  {
    for (USHORT j = 0; j < w; j++)
    {
      cout << "* ";
    }
    cout << endl;
  }
}
```

Listing 2: rectangulo.cpp

```
#include "rectangulo.h"
#include <memory>
```

```cpp

using namespace std;

int main()
{
  auto R1 = make_unique<Rectangulo>();
  cout << "\n[+] R1 - Draw(): " << endl;
  R1->Draw();

  auto R2 = make_unique<Rectangulo>(5, 3);
  cout << "\n[+] R2 - Draw(): " << endl;
  R2->Draw();

  cout << "\n[+] R1 - Draw(8, 5): " << endl;
  R1->Draw(8, 5);

  auto R3 = make_unique<Rectangulo>(*R1);
  cout << "\n[+] R3 - Draw(): " << endl;
  R3->Draw();

  auto R4 = make_unique<Rectangulo>(*R2);
  cout << "\n[+] R4 - Draw(): " << endl;
  R4->Draw();

  return 0;
}
```

Listing 3: main.cpp

Ejecución del ejercicio

```
> .\main.exe

[+] R1 - Draw():
*

[+] R2 - Draw():
* * * * *
* * * * *
* * * * *

[+] R1 - Draw(8, 5):
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *

[+] R3 - Draw():
*

[+] R4 - Draw():
* * * * *
* * * * *
* * * * *
```

Análisis del código:

El código implementa la clase `Rectangulo`, que usa `unique_ptr` para gestionar dinámicamente la memoria de atributos como `width`, `height`, `temperatura`, `presion` y `humedad`. Se incluyen tres constructores y un método para dibujar el rectángulo.

Se ofrecen tres constructores:

- `Rectangulo()` - Crea un rectángulo de 1x1 con valores predeterminados.
- `Rectangulo(USHORT w, USHORT h)` - Inicializa un rectángulo con dimensiones personalizadas.
- `Rectangulo(const Rectangulo &R)` - Copia profunda de otro objeto `Rectangulo`.

`Draw()` asigna valores a `temperatura`, `presion` y `humedad` antes de llamar a la versión sobrecargada que dibuja el rectángulo con las dimensiones proporcionadas.

El uso de `unique_ptr` asegura la gestión eficiente de memoria, liberándola automáticamente cuando los objetos son destruidos, evitando fugas de memoria.

2.2. Ejercicio 2

- Modificar el ejercicio 02 (time) para que funcione en memoria dinámica. Los miembros de la clase deben ser punteros, las instancias en el main deben cargarse en memoria dinámica, el constructor debe crear la instancia en la memoria dinámica. Utilizar punteros inteligentes.

```cpp
#ifndef TIME_H
#define TIME_H
#include <memory>
#include <iostream>

using namespace std;
typedef unsigned short int USHORT;

class Time
{
private:
  unique_ptr<USHORT> hour;
  unique_ptr<USHORT> minute;
  unique_ptr<USHORT> second;

public:
  Time(const USHORT h = 0, const USHORT m = 0, const USHORT s = 0);
  Time(const Time &t);
  ~Time() {}

  void setTime(const USHORT h, const USHORT m, const USHORT s);
  void printTime() const;
  bool equals(const Time &T) const;
};

#endif
```

Listing 4: time.h

```cpp
#include "Time.h"
#include <iomanip>

using namespace std;

Time::Time(const USHORT h, const USHORT m, const USHORT s) : hour(make_unique<USHORT>(h)), minute(make_u

Time::Time(const Time &T) : hour(make_unique<USHORT>(*T.hour)), minute(make_unique<USHORT>(*T.minute)),

void Time::setTime(const USHORT h, const USHORT m, const USHORT s)
{
  *hour = h;
  *minute = m;
  *second = s;
}

void Time::printTime() const
{
  cout << setfill('0') << setw(2) << *hour << ":"
       << setfill('0') << setw(2) << *minute << ":"
       << setfill('0') << setw(2) << *second << endl;
}

bool Time::equals(const Time &T) const
{
  return (*hour == *T.hour) && (*minute == *T.minute) && (*second == *T.second);
}
```

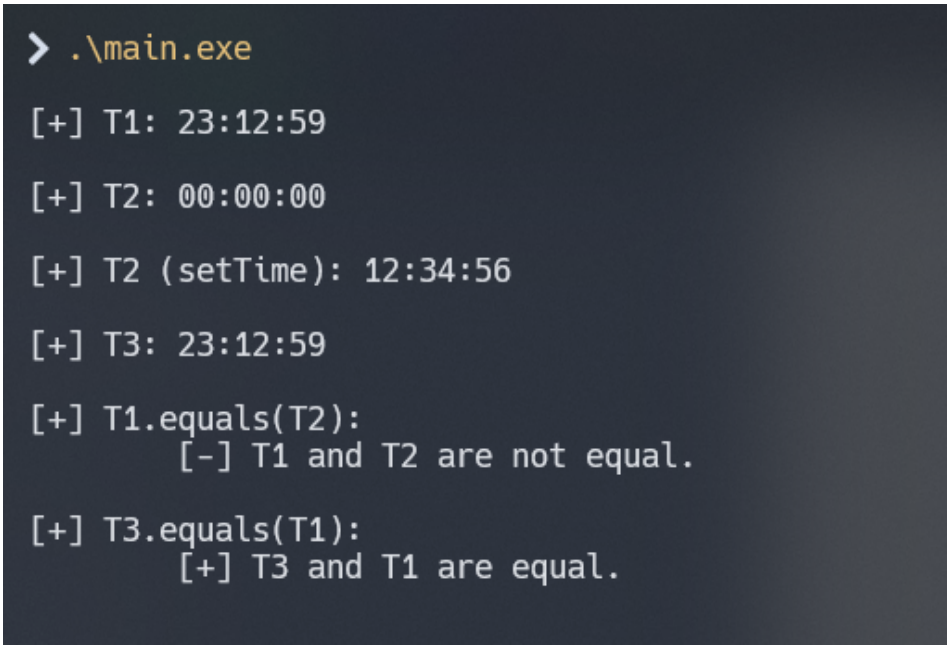
Listing 5: time.cpp

```cpp
#include "Time.h"

using namespace std;

int main()
```

```cpp
{
  auto T1 = make_unique<Time>(23, 12, 59);
  cout << "\n[+] T1: ";
  T1->printTime();

  auto T2 = make_unique<Time>();
  cout << "\n[+] T2: ";
  T2->printTime();

  T2->setTime(12, 34, 56);
  cout << "\n[+] T2 (setTime): ";
  T2->printTime();

  auto T3 = make_unique<Time>(*T1);
  cout << "\n[+] T3: ";
  T1->printTime();

  cout << "\n[+] T1.equals(T2): " << endl;
  if (T1->equals(*T2))
    cout << "\t[+] T1 and T2 are equal." << endl;
  else
    cout << "\t[-] T1 and T2 are not equal." << endl;

  cout << "\n[+] T3.equals(T1): " << endl;
  if (T3->equals(*T1))
    cout << "\t[+] T3 and T1 are equal." << endl;
  else
    cout << "\t[-] T3 and T1 are not equal." << endl;

  cout << "\n\n";
  return 0;
}
```

Listing 6: main.cpp

Ejecución del ejercicio



Análisis del código:

El código implementa una clase `Time` que gestiona la hora usando punteros inteligentes `unique_ptr` para las horas, minutos y segundos, asegurando una correcta gestión de la memoria.

La clase define dos constructores:

- `Time(USHORT h=0, USHORT m=0, USHORT s=0)` - Inicializa la hora con los valores proporcionados o por defecto a 00:00:00.
- `Time(const Time &T)` - Constructor de copia que realiza una copia profunda de otro objeto `Time`.
- `void setTime(USHORT h, USHORT m, USHORT s)` - Establece una nueva hora.
- `void printTime()` - Imprime la hora en formato `hh:mm:ss`, con ceros a la izquierda.
- `bool equals(const Time &T)` - Compara si dos objetos `Time` tienen la misma hora.

El uso de `unique_ptr` para gestionar los atributos garantiza que no haya fugas de memoria, ya que estos punteros gestionan automáticamente el ciclo de vida de la memoria asignada.

En `main()`, se crean varios objetos `Time` utilizando tanto el constructor por defecto como el constructor parametrizado. Se comparan las horas de los objetos usando el método `equals()` y se imprimen los resultados de la comparación.

2.3. Ejercicio 3

- Modificar el ejercicio 02 (time) para que funcione en memoria dinámica. Los miembros de la clase deben ser punteros, las instancias en el main deben cargarse en memoria dinámica, el constructor debe crear la instancia en la memoria dinámica. Utilizar punteros inteligentes.

```
#ifndef NODO_H
#define NODO_H

#include <iostream>

using namespace std;

class Nodo
{
public:
  int valor;
  Nodo *siguiente;
  Nodo *anterior;

  Nodo(int val) : valor(val), siguiente(nullptr), anterior(nullptr) {}

  ~Nodo()
  {
    cout << "\nNodo eliminado: " << valor << endl;
  }
};

#endif
```

Listing 7: Nodo.h

```cpp
#ifndef LISTA_H
#define LISTA_H

#include "Nodo.h"

class Lista
{
private:
  Nodo *cabeza;
  Nodo *cola;

public:
  Lista();
  ~Lista();

  void insertarInicio(int valor);
  void insertarFinal(int valor);

  void eliminarInicio();
  void eliminarFinal();

  void imprimeDesdeAdelante() const;
  void imprimeDesdeAtras() const;

  Nodo *buscarDesdeAdelante(int valor) const;
  Nodo *buscarDesdeAtras(int valor) const;

  void destruyeLista();
};

#endif
```

Listing 8: Lista.h

```cpp
#include "Lista.h"

Lista::Lista() : cabeza(nullptr), cola(nullptr)
{
  cout << "\n[+] Constructor por defecto" << endl;
}

Lista::~Lista()
{
  destruyeLista();
  cout << endl;
}

void Lista::insertarInicio(int valor)
{
  Nodo *nuevoNodo = new Nodo(valor);

  if (cabeza == nullptr)
  {
    cabeza = cola = nuevoNodo;
  }
  else
  {
    nuevoNodo->siguiente = cabeza;
    cabeza->anterior = nuevoNodo;
    cabeza = nuevoNodo;
  }
}

void Lista::insertarFinal(int valor)
```

```
{
  Nodo *nuevoNodo = new Nodo(valor);

  if (cola == nullptr)
  {
    cabeza = cola = nuevoNodo;
  }
  else
  {
    cola->siguiente = nuevoNodo;
    nuevoNodo->anterior = cola;
    cola = nuevoNodo;
  }
}

void Lista::eliminarInicio()
{
  if (cabeza == nullptr)
  {
    cout << "\n[-] No se puede eliminar, la lista est  vac a." << endl;
    return;
  }

  Nodo *nodoAEliminar = cabeza;

  if (cabeza == cola)
  {
    cabeza = cola = nullptr;
  }
  else
  {
    cabeza = cabeza->siguiente;
    cabeza->anterior = nullptr;
  }

  delete nodoAEliminar;
}

void Lista::eliminarFinal()
{
  if (cola == nullptr)
  {
    cout << "\n[-] No se puede eliminar, la lista est  vac a." << endl;
    return;
  }

  Nodo *nodoAEliminar = cola;

  if (cabeza == cola)
  {
    cabeza = cola = nullptr;
  }
  else
  {
    cola = cola->anterior;
    cola->siguiente = nullptr;
  }

  delete nodoAEliminar;
}

void Lista::imprimeDesdeAdelante() const
{
  Nodo *actual = cabeza;
  cout;
```

```
96    while (actual != nullptr)
97    {
98      cout << actual->valor << " ";
99      actual = actual->siguiente;
100   }
101   cout << endl;
102 }
103
104 void Lista::imprimeDesdeAtras() const
105 {
106   Nodo *actual = cola;
107   cout;
108   while (actual != nullptr)
109   {
110     cout << actual->valor << " ";
111     actual = actual->anterior;
112   }
113   cout << endl;
114 }
115
116 Nodo *Lista::buscarDesdeAdelante(int valor) const
117 {
118   Nodo *actual = cabeza;
119   while (actual != nullptr)
120   {
121     if (actual->valor == valor)
122       return actual;
123     actual = actual->siguiente;
124   }
125   return nullptr;
126 }
127
128 Nodo *Lista::buscarDesdeAtras(int valor) const
129 {
130   Nodo *actual = cola;
131   while (actual != nullptr)
132   {
133     if (actual->valor == valor)
134       return actual;
135     actual = actual->anterior;
136   }
137   return nullptr;
138 }
139
140 void Lista::destruyeLista()
141 {
142   Nodo *actual = cabeza;
143   while (actual != nullptr)
144   {
145     Nodo *siguiente = actual->siguiente;
146     delete actual;
147     actual = siguiente;
148   }
149   cabeza = cola = nullptr;
150 }
```

Listing 9: Lista.cpp

```
1  #include "Lista.h"
2
3  int main()
4  {
5    Lista lista;
6
7    lista.insertarInicio(10);
```

```
  lista.insertarFinal(20);
  lista.insertarInicio(5);
  lista.insertarInicio(17);
  lista.insertarFinal(30);
  lista.insertarFinal(25);

  cout << "\n[+] Lista desde el inicio: ";
  lista.imprimeDesdeAdelante();

  cout << "\n[+] Lista desde el final: ";
  lista.imprimeDesdeAtras();

  lista.eliminarInicio();
  cout << "[+] Lista despues de eliminar el inicio: ";
  lista.imprimeDesdeAdelante();

  lista.eliminarFinal();
  cout << "[+] Lista despues de eliminar el final: ";
  lista.imprimeDesdeAdelante();

  int valorBuscado = 20;
  Nodo *encontrado = lista.buscarDesdeAdelante(valorBuscado);
  if (encontrado)
    cout << "\nElemento " << valorBuscado << " encontrado desde adelante." << endl;
  else
    cout << "\nElemento " << valorBuscado << " no encontrado desde adelante." << endl;

  encontrado = lista.buscarDesdeAtras(valorBuscado);
  if (encontrado)
    cout << "\nElemento " << valorBuscado << " encontrado desde atras." << endl;
  else
    cout << "\nElemento " << valorBuscado << " no encontrado desde atras." << endl;

  cout << "\n\n[-] Destruyendo la lista..." << endl;

  return 0;
}
```

Listing 10: main.cpp

Ejecución del ejercicio

```
> .\main.exe

[+] Constructor por defecto

[+] Lista desde el inicio: 17 5 10 20 30 25

[+] Lista desde el final: 25 30 20 10 5 17

Nodo eliminado: 17
[+] Lista despues de eliminar el inicio: 5 10 20 30 25

Nodo eliminado: 25
[+] Lista despues de eliminar el final: 5 10 20 30

Elemento 20 encontrado desde adelante.

Elemento 20 encontrado desde atras.


[-] Destruyendo la lista...

Nodo eliminado: 5

Nodo eliminado: 10

Nodo eliminado: 20

Nodo eliminado: 30
```

Análisis del código:

El código implementa una lista doblemente enlazada mediante las clases `Nodo` y `Lista`, proporcionando métodos para insertar, eliminar, buscar e imprimir elementos desde ambos extremos de la lista.

La clase `Nodo` representa un nodo de la lista, compuesto por:

- `int valor` - Almacena el valor del nodo.
- `Nodo *siguiente` - Apunta al siguiente nodo.
- `Nodo *anterior` - Apunta al nodo anterior.

El constructor inicializa los punteros a `nullptr` y el destructor imprime un mensaje cuando se elimina un nodo.

La clase `Lista` gestiona las operaciones de la lista mediante los punteros `cabeza` y `cola`, que apuntan al primer y último nodo, respectivamente.

- `insertarInicio(int valor)` - Inserta un nuevo nodo al principio de la lista.
- `insertarFinal(int valor)` - Inserta un nuevo nodo al final de la lista.
- `eliminarInicio()` - Elimina el nodo al principio de la lista.
- `eliminarFinal()` - Elimina el nodo al final de la lista.

Ambos métodos manejan correctamente la eliminación de nodos, ajustando los punteros `cabeza` y `cola` cuando es necesario.

- `buscarDesdeAdelante(int valor)` y `buscarDesdeAtras(int valor)` - Buscan un nodo desde el inicio o el final de la lista.
- `imprimeDesdeAdelante()` e `imprimeDesdeAtras()` - Imprimen los valores de los nodos desde el inicio o el final de la lista.

El método `destruyeLista()` recorre toda la lista, eliminando los nodos para evitar fugas de memoria.

Se realizó una implementación robusta de una lista doblemente enlazada, con métodos claros para su manipulación, incluyendo inserciones, eliminaciones y búsqueda eficiente en ambos sentidos de la lista.

2.4. Ejercicio 4

- Implementar en C++ un Binary Expression Tree que pueda resolver operaciones matemáticas suma y multiplicación (usar clases y punteros).

```cpp
#ifndef NODO_H
#define NODO_H

class Nodo
{
public:
  virtual int evaluar() const = 0;
  virtual void imprimir() const = 0;
  virtual ~Nodo() = default;
};

#endif
```

Listing 11: Nodo.h

```cpp
#ifndef NODONUMERO_H
#define NODONUMERO_H

#include "Nodo.h"
#include <iostream>

class NodoNumero : public Nodo
{
private:
  int valor;

public:
  NodoNumero(int v);
  int evaluar() const override;
  void imprimir() const override;
};

#endif
```

Listing 12: NodoNumero.h

```cpp
#include "NodoNumero.h"

NodoNumero::NodoNumero(int v) : valor(v) {}

int NodoNumero::evaluar() const
{
  return valor;
}

void NodoNumero::imprimir() const
{
  std::cout << valor;
}
```

Listing 13: NodoNumero.cpp

```h
#ifndef NODOOPERACION_H
#define NODOOPERACION_H

#include "Nodo.h"
#include <memory>
#include <iostream>

class NodoOperacion : public Nodo
{
private:
  char operador;
  std::unique_ptr<Nodo> hijoIzquierdo;
  std::unique_ptr<Nodo> hijoDerecho;

public:
  NodoOperacion(char op, std::unique_ptr<Nodo> izq, std::unique_ptr<Nodo> der);
  int evaluar() const override;
  void imprimir() const override;
};

#endif
```

Listing 14: NodoOperacion.h

```cpp
#include "NodoNumero.h"

NodoNumero::NodoNumero(int v) : valor(v) {}

int NodoNumero::evaluar() const
{
  return valor;
}

void NodoNumero::imprimir() const
{
  std::cout << valor;
}
```

Listing 15: NodoNumero.cpp

```h
#ifndef NODOOPERACION_H
#define NODOOPERACION_H

#include "Nodo.h"
#include <memory>
#include <iostream>

class NodoOperacion : public Nodo
{
private:
  char operador;
  std::unique_ptr<Nodo> hijoIzquierdo;
  std::unique_ptr<Nodo> hijoDerecho;

public:
  NodoOperacion(char op, std::unique_ptr<Nodo> izq, std::unique_ptr<Nodo> der);
  int evaluar() const override;
  void imprimir() const override;
};

#endif
```

Listing 16: NodoOperacion.h

```cpp
#include "Analizador.h"
#include <cctype>

std::unique_ptr<Nodo> Analizador::analizar(const std::string &expresion)
{
  pos = 0;
  return parseExpresion(expresion);
}

std::unique_ptr<Nodo> Analizador::parseExpresion(const std::string &expresion)
{
  auto nodoIzq = parseTermino(expresion);

  while (pos < expresion.size() && (expresion[pos] == '+' || expresion[pos] == '-'))
  {
    char operador = expresion[pos++];
    auto nodoDer = parseTermino(expresion);
    nodoIzq = std::make_unique<NodoOperacion>(operador, std::move(nodoIzq), std::move(nodoDer));
  }

  return nodoIzq;
}

std::unique_ptr<Nodo> Analizador::parseTermino(const std::string &expresion)
{
  auto nodo = parseNumero(expresion);

  while (pos < expresion.size() && (expresion[pos] == '*' || expresion[pos] == '/'))
  {
    char operador = expresion[pos++];
    auto nodoDer = parseNumero(expresion);
    nodo = std::make_unique<NodoOperacion>(operador, std::move(nodo), std::move(nodoDer));
  }

  return nodo;
}

std::unique_ptr<Nodo> Analizador::parseNumero(const std::string &expresion)
{
  std::string numeroStr;

  while (pos < expresion.size() && std::isdigit(expresion[pos]))
  {
    numeroStr += expresion[pos++];
  }

  return std::make_unique<NodoNumero>(std::stoi(numeroStr));
}
```

Listing 17: analizador.cpp

```cpp
#include <iostream>
#include "Analizador.h"

using namespace std;

int main()
{
  string expresion;

  cout << "\n[+] Ingrese la expresion: ";
  cin >> expresion;

  Analizador analizador;
```

```
14    unique_ptr<Nodo> raiz = analizador.analizar(expresion);
15
16    cout << "\n[+] Resultado: " << raiz->evaluar() << endl;
17
18    cout << "\n[+] Arbol de expresion: ";
19    raiz->imprimir();
20
21    cout << "\n"
22         << endl;
23
24    return 0;
25  }
```

Listing 18: main.cpp

Ejecución del ejercicio

```
> .\main.exe
[+] Ingrese la expresion: 54+5*34+1*2
[+] Resultado: 226
[+] Arbol de expresion: ((54 + (5 * 34)) + (1 * 2))
```

```
> .\main.exe
[+] Ingrese la expresion: 78-48+5*21-456
[+] Resultado: -321
[+] Arbol de expresion: (((78 - 48) + (5 * 21)) - 456)
```

```
> .\main.exe
[+] Ingrese la expresion: 45-5/5*3-3
[+] Resultado: 39
[+] Arbol de expresion: ((45 - ((5 / 5) * 3)) - 3)
```

Análisis del código:

El programa desarrollado crea un Árbol de Expresiones Binarias, que permite evaluar operaciones matemáticas básicas (`+`, `-`, `*`, `/`) y generar una representación en notación infija de la expresión. Se compone de las siguientes clases y componentes principales:

La clase abstracta `Nodo` define la interfaz básica para los nodos del árbol, que incluye dos métodos virtuales puros:

- `evaluar()` - Devuelve el valor calculado del nodo.

- `imprimir()` - Imprime el contenido del nodo.

Esta clase sirve como base para los nodos de tipo número y operación.

Derivada de la clase `Nodo`, esta clase representa un nodo hoja que contiene un valor numérico. Implementa los métodos `evaluar()` para devolver el valor almacenado y `imprimir()` para mostrarlo.

Esta clase también hereda de `Nodo`, y se utiliza para representar operaciones binarias (`+`, `-`, `*`, `/`). Contiene:

- `operador` - Un carácter que representa la operación.
- `hijoIzquierdo` y `hijoDerecho` - Punteros únicos a los nodos hijos, que pueden ser otros nodos de operación o número.

El método `evaluar()` realiza la operación entre los valores evaluados de los hijos, mientras que `imprimir()` muestra la expresión en notación infija, envolviendo la operación entre paréntesis.

El `Analizador` procesa una expresión matemática representada como cadena y construye el árbol de expresiones correspondiente. Para ello, utiliza tres funciones:

- `parseExpresion()` - Procesa términos separados por sumas o restas.
- `parseTermino()` - Procesa factores separados por multiplicaciones o divisiones.
- `parseNumero()` - Extrae un número de la cadena de entrada.

El árbol se construye recursivamente al identificar los operadores y agrupar los operandos en subárboles.

El programa principal solicita al usuario que ingrese una expresión. El analizador crea el árbol de la expresión, lo evalúa y luego imprime la expresión en notación infija.

```
[+] Ingrese la expresion: 3+5*2
[+] Resultado: 13
[+] Arbol de expresion: (3 + (5 * 2))
```

3. Cuestionario

3.1. Buscar y explicar sobre sobrecarga versus polimorfismo en C++. Proponer ejemplos.

3.1.1. Sobrecarga

La sobrecarga de funciones permite declarar múltiples funciones con el mismo nombre pero diferentes parámetros, y se resuelve en tiempo de compilación. Ejemplo:

```
#include <iostream>

void imprimir(int x) {
    std::cout << "Entero: " << x << std::endl;
}
void imprimir(double x) {
    std::cout << "Decimal: " << x << std::endl;
}

int main() {
    imprimir(5);    // Llama a la función con int
    imprimir(3.14); // Llama a la función con double
}
```

3.1.2. Polimorfismo

El polimorfismo se refiere a la capacidad de clases derivadas de sobrescribir métodos de una clase base, y se resuelve en tiempo de ejecución, usando funciones virtuales. Ejemplo:

```
#include <iostream>

class Animal {
public:
    virtual void sonido() const { std::cout << "Sonido genérico\n"; }
};

class Perro : public Animal {
public:
    void sonido() const override { std::cout << "El perro ladra\n"; }
};

int main() {
    Animal* a = new Perro();
    a->sonido();  // Llama a la versión de Perro
    delete a;
}
```

3.1.3. Diferencias

- La sobrecarga ocurre en tiempo de compilación y se basa en la firma de la función. - El polimorfismo ocurre en tiempo de ejecución y depende del tipo real del objeto.

3.2. Buscar información de punteros inteligente. Poner ejemplos.

3.2.1. std::unique_ptr

Es un puntero inteligente que posee un único propietario. El recurso se libera cuando el puntero es destruido o transferido. Ejemplo:

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> ptr = std::make_unique<int>(42);
    std::cout << *ptr << std::endl;
}
```

3.2.2. std::shared_ptr

Permite que varios punteros compartan la propiedad de un recurso. Lleva un contador de referencias y libera el recurso cuando el último puntero se destruye. Ejemplo:

```
#include <iostream>
#include <memory>

int main() {
```

```
    std::shared_ptr<int> p1 = std::make_shared<int>(42);
    std::shared_ptr<int> p2 = p1;  // Comparte el recurso
    std::cout << *p2 << std::endl;
}
```

3.2.3. std::weak_ptr

No incrementa el contador de referencias. Se usa para evitar ciclos de referencias. Ejemplo:

```
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> sp = std::make_shared<int>(42);
    std::weak_ptr<int> wp = sp;  // No afecta el conteo de referencias

    if (auto sp2 = wp.lock()) {  // Accede al recurso
        std::cout << *sp2 << std::endl;
    }
}
```