Object Oriented Design & Programming Tutorial
Abstract Classes and Pure Virtual Functions

Some background on expressions:

- An `Expression` encapsulates an arithmetic expression. In this tutorial, they involve only integers and two operations on them: addition and multiplication.
- A `Number` is a kind of `Expression` that encapsulates an integer value.
- An `LBinary` is a kind of `Expression` that represents a left-associative binary operation between two sub-expressions. It has a left operand, an operator symbol, and a right operand. (An operation # is left-associative if e_1 # e_2 # e_3 means (e_1 # e_2) # e_3.)
- `Addition` and `Multiplication` are two kinds of `LBinary`. The symbol of an `Addition` is `"+"`, and the symbol of a `Multiplication` is `"*"`.
- Every expression can be evaluated. The evaluation result is the same as that of the arithmetic expression it represents.

You are given an implementation below. Notice how the abstract class `LBinary` evaluates itself using Template Method Pattern.

Task

Add a global operator overload:

```
ostream &operator <<(ostream &o, const Expression &e)
```

to the program, which outputs the textual representation of the arithmetic expression, followed by `" = "`, followed by the result of evaluation.

To prevent incorrect interpretation, sub-expressions may need to be bracketed. E.g. the output "`1+2*3`" is wrong if the expression means "`(1+2)*(3)`"; the output "`1+2+3`" is also wrong for an expression that means "`(1)+(2+3)`". You can use as many pairs of brackets as you want, as long as the output describes the same structure as stored in the object `e`.

You can add any number of global functions, new classes, and new members of existing classes to get you to the goal, as long as you don't change the existing members in the existing classes.

Hints:

- The best solution defines only one `operator <<` and no more global functions.
- Check out how `LBinary::value()` delegates the calculation process to the two operands, and apply the same structure to your outputting function.

Extra Challenge

If you like challenges, try to use just enough brackets to show the correct precedence and associativity:

- "(1)+(2*3)" should be "1+2*3"; and "(1+2)+3" should be "1+2+3". But "1+(2+3)" should NOT become "1+2+3" due to the left associativity of +.
- A negative number alone or as the left operand of an addition should be printed with no brackets, e.g. "-1" and "-1+2"; but for all other cases it should be bracketed, e.g. "3+(-4)" and "(-5)*6".
- The codes you add to the program above should require no further adaptation, if later we want to add these three expression types: Subtraction (e.g. "-1-2"), Division (e.g. "6/3") and Exponentiation (e.g. "2^4", meaning 2^4. Note: ^ binds tighter than * and /, e.g. "3*2^4" means 3 * 2^4).

Hints for the extra challenge:

- Every expression has a precedence ranking. When outputting a left associative binary expression, the precedence ranking of that expression should be compared with those of its two operands, to decide if bracketing is necessary.
- The superclass of Addition and Multiplication is called **L**Binary for a reason.
- The precedence ranking of a Number is slightly trickier than that of an Addition or Multiplication.
- To bracket or not to bracket; that is not the question – the real question is who should make the decision

```
class Expression
{
public:
   virtual int value() const = 0;
};

class Number : public Expression
{
   int _value;
public:
   Number(int value) : _value(value) {}
   int value() const {
      return _value;
   }
};

class LBinary : public Expression
{
   Expression &_left, &_right;
protected:
   virtual int calculate(int v1, int v2) const = 0;
public:
   LBinary(Expression &left, Expression &right) : _left(left), _right(right) {}
   Expression &left() const {
      return _left;
   }
   Expression &right() const {
      return _right;
   }
   virtual const char *symbol() const = 0;
   virtual int value() const {
      int left_value = _left.value();
      int right_value = _right.value();
      int result = calculate(left_value, right_value);
      return result;
      // one-line version: return calculate(_left.value(), _right.value());
   }
};

class Addition : public LBinary
{
protected:
   virtual int calculate(int v1, int v2) const {
      return v1 + v2;
   }
public:
   Addition(Expression &left, Expression &right) : LBinary(left, right) {}
   virtual const char *symbol() const {
      return "+";
   }
};
```

```
class Multiplication : public LBinary
{
protected:
    virtual int calculate(int v1, int v2) const {
        return v1 * v2;
    }
public:
    Multiplication(Expression &l, Expression &r) : LBinary(l, r) {}
    virtual const char *symbol() const {
        return "*";
    }
};
```

A sample main function is provided below:

```
int main() {
    Number n2(2), n3(3), n_1(-1);
    cout << n_1 << endl;
    // SIMPLE OUTPUT : -1 = -1
    // MINIMUM OUTPUT: -1 = -1

    Addition a1(n2, n_1);
    cout << a1 << endl;
    // SIMPLE OUTPUT : (2)+(-1) = 1
    // MINIMUM OUTPUT: 2+(-1) = 1

    Addition a2(n_1, n3);
    cout << a2 << endl;
    // SIMPLE OUTPUT : (-1)+(3) = 2
    // MINIMUM OUTPUT: -1+3 = 2

    Multiplication m1(n3, a1);
    cout << m1 << endl;
    // SIMPLE OUTPUT : (3)*((2)+(-1)) = 3
    // MINIMUM OUTPUT: 3*(2+(-1)) = 3

    Addition a3(m1, a1);
    cout << a3 << endl;
    // SIMPLE OUTPUT : ((3)*(2+(-1)))+(2+(-1)) = 4
    // MINIMUM OUTPUT: 3*(2+(-1))+(2+(-1)) = 4

    Addition a4(a1, m1);
    cout << a4 << endl;
    // SIMPLE OUTPUT : ((2)+(-1))+((3)*(2+(-1))) = 4
    // MINIMUM OUTPUT: 2+(-1)+3*(2+(-1)) = 4
}
```

After the tutorial session, take this exercise to the lab and do some actual compiling. You will learn more through trial and error, than literally “writing” programs with pen and paper.