Abstract Classes and Pure Virtual Functions

1 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 e1 "# e2 # e3" means "(e1 # e2) # e3".
- 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.

2 Task

Add a global operator overload to the given implementation below. Notice how the abstract class *LBinary* evaluates itself using Template Method Pattern. Add the 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 its evaluation. To prevent incorrect interpretation, sub-expressions may need to be bracketed. For example, 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 do not change the existing members in the existing classes.

Hints:

- A good 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.

3 Extra Challenge

For an extra challenge, 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 16). Note: Exponents binds tighter than $*$ and $/$, e.g. $3*2^4$ means $3*16$).

Hints:

- 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 *LBinary* for a reason.
- The precedence ranking of a *Number* is slightly trickier than that of an *Addition* or *Multiplication*.
- Think about where the decision on whether to bracket or not should me made.

4 Files

Skeleton files (main.cpp and expression.cpp) are provided on CATE.