BLG252E OBJECT-ORIENTED PROGRAMMING

HW#2

Due Date: 22.04.2019

In this homework, you are asked to place a couple of mathematical operators in a grid and perform some operations on the placed operators. You are given the following Class declarations:

Operator Class

```
class Operator {
   int center_x;
   int center_y;
   int op_size;

public:
   Operator(int x, int y, int size);

   void reset (int new_x, int new_y, int new_size);

   void set_x( int new_x );
   int  get_x();

   void set_y( int new_y );
   int  get_y();

   void set_size( int new_size );
   int  get_size();
};
```

Operator class defines a generic operator that is located on a (**center_x**, **center_y**) position on a grid with size **op_size** (you can assume that **op_size** is a number in the interval **[1..9]**). It has accessor (getter) and mutator (setter) methods for all the instance variables. In addition, it is possible to reset all these variables together with **reset()** method.

ArithmeticOperator Class

```
class ArithmeticOperator: public Operator {
   char sign;

public:
   ArithmeticOperator(int x, int y, int size, char sign);

   char get_sign();

   // Prints out operator's center location, size, and sign character
   void print_operator();
};
```

ArithmeticOperator class adds a **sign** field to specify the operator. The sign can take the values **+**, **-**, **x**, or **/**. If a value other than this set is given as the sign parameter in the constructor, then the message

SIGN parameter is invalid!

must be printed out.

Since we don't want to change a **sign** after instantiation of an object, we don't supply a mutator method for it. However, we have the **get_sign()** accessor method.

print_operator() method prints out the message:

ARITHMETIC_OPERATOR[**sign**], CENTER_LOCATION[**center_x**,**center_y**], SIZE[**op_size**]

where **sign** is the sign of the operator, **center_x** and **center_y** are center location of the operator (defined in the superclass **Operator**), and **op_size** is the size of the operator (defined in the superclass **Operator**).

OperatorGrid Class

```
class OperatorGrid {
   int grid_rows;
   int grid_cols;
   char **grid;

   int num_operators;
   ArithmeticOperator *operators[MAX_OPERATOR_SIZE];

public:
   OperatorGrid(int rows, int cols);
   ~OperatorGrid();

   bool place_operator (ArithmeticOperator *);
   bool move_operator  (int x, int y, char direction, int move_by);
   void print_operators();
};
```

The constructor **OperatorGrid(int rows, int cols)** creates an **OperatorGrid** instance that consists of **rows grid_rows** and **columns grid_columns**, as shown in Figure 1. Note that **row/column numbers start from 1 and not from 0**.

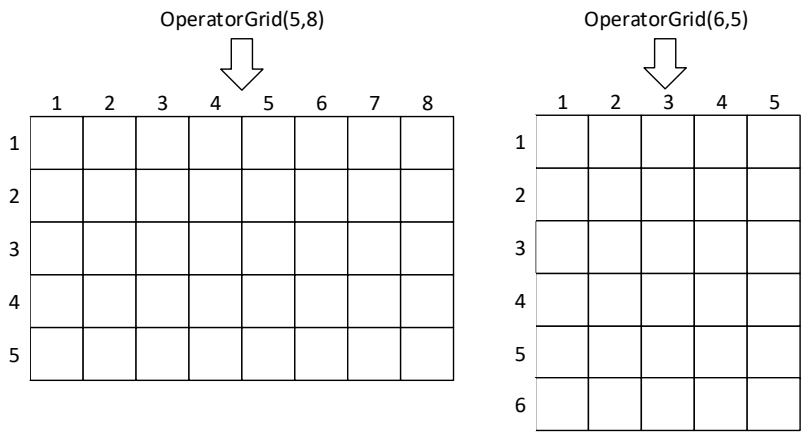

Figure 1: Two grids with sizes specified in the OperatorGrid constructor

OperatorGrid **destructor** must give back (i.e., delete) the dynamically allocated memory locations for **grid** and **operators**. It prints out the following message:

DESTRUCTOR: GIVE BACK[**rows**,**cols**] chars.
DESTRUCTOR: GIVE BACK[**num_operators**] Operators.

where **rows** is **grid_rows**, **cols** is **grid_cols**, and **num_operators** is the **num_operators** instance variable of the OperatorGrid instance that is being destructed.

place_operator(ArithmeticOperator *operator) method places a dynamically created **ArithmeticOperator** instance on the grid. Figure 2 shows placement of some instances with **op_size**=1. Please note that, each operator is placed on the grid so that the complete shape reflects the type of the operator which is determined by looking at the **sign** instance variable. For example,

a. **Leftmost sample** places a **+** (**sign='+'**) character on the center location **[2,2]** and adds one (**op_size=1**) **+** character to right, left, upper, and bottom of the center.
b. **Rightmost sample** places a **/** (**sign='/'**) character on the center location **[4,2]** and adds one (**op_size=1**) **/** character to upper-right and bottom-left of the center.

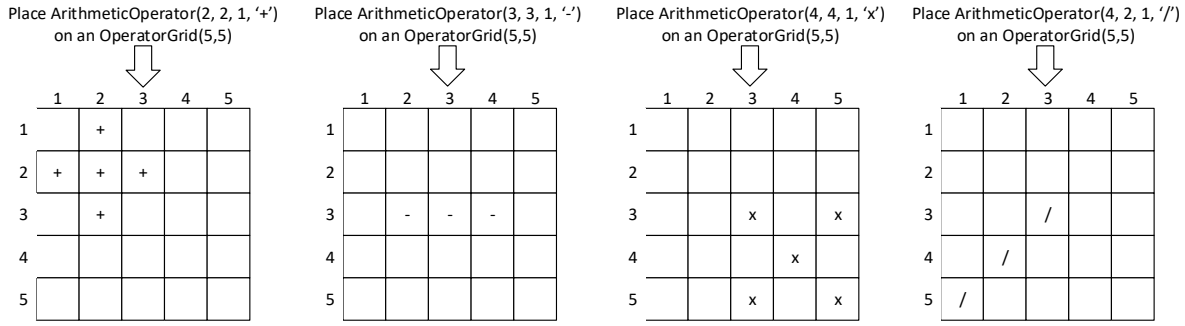

Figure 2: Placement of unit operators (i.e., operators with op_size=1) on 5x5 grids, centered on different notations.

Similarly, Figure 3 shows **ArithmeticOperator** instances with **op_size=**2 that are placed on the center of 5x5 grids.

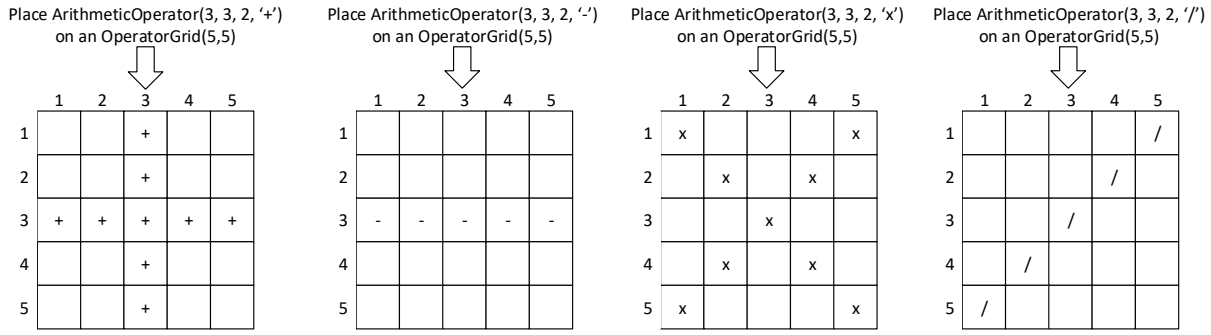

Figure 3: Placement of operators with op_size=2 on 5x5 grids, operators are centered on the grid (i.e., center location=[3,3]).

When you place an operator, you have to check whether the operator fits into the grid or not. There is a **BORDER ERROR** when **at least one cell** of the operator is placed outside of the grid. Figure 4 shows some example **BORDER ERROR**s (i.e., operator overflows outside of the grid).

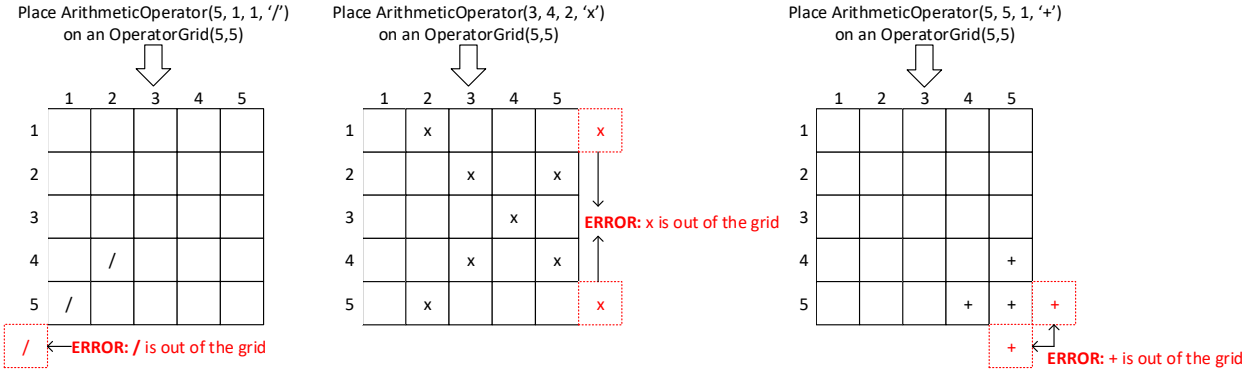

Figure 4: Some BORDER ERROR examples for invalid operator placements.

When you place an operator, you have to check whether the cells were occupied by another previously placed operator or not. When **at least one cell** is occupied by a previously placed operator, there is a **CONFLICT ERROR**. Figure 5 examplifies confict errors.

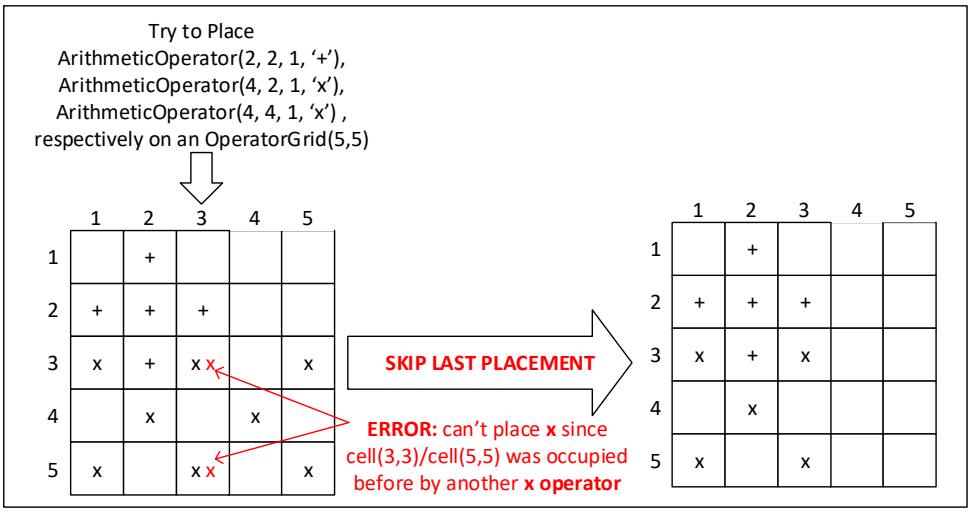

Figure 5: Example of CONFLICT ERRORs

When an error (BORDER **and/or** CONFLICT) occured in placing an operator on a grid, you skip the placement. If the placement is successful, then add the operator into the **operators** instance variable of the **OperatorGrid** instance. This variable must contain the operators **in their insertion order**.

place_operator() method returns **true** if the placement is successful. Otherwise, it returns **false**. In addition, at the end of the **place_operator()** method:

a) Print out following message if the operator is successfully placed (see Figure 2 and Figure 3) on the **OperatorGrid** instance:

SUCCESS: Operator **sign** with size **op_size** is placed on (**center_x**,**center_y**).

where **sign** is the sign of the operator, **center_x** and **center_y** are center location of the operator (defined in the superclass **Operator**), and **op_size** is the size of the operator (defined in the superclass **Operator**).

b) If there is a **BORDER ERROR** (see Figure 4), then omit the placement and print out the following message:

BORDER ERROR: Operator **sign** with size **op_size** can not be placed on (**center_x**,**center_y**).

where **sign** is the sign of the operator, **center_x** and **center_y** are center location of the operator (defined in the superclass **Operator**), and **op_size** is the size of the operator (defined in the superclass **Operator**).

c) If there is a **CONFLICT ERROR** (see Figure 5), then omit the placement and print out the following message:

CONFLICT ERROR: Operator **sign** with size **op_size** can not be placed on (**center_x**,**center_y**).

where **sign** is the sign of the operator, **center_x** and **center_y** are center location of the operator (defined in the superclass **Operator**), and **op_size** is the size of the operator (defined in the superclass **Operator**).

d) If there are **both BORDER and CONFLICT ERRORS**, then omit the placement and print out the following messages **in order**:

BORDER ERROR: Operator **sign** with size **op_size** can not be placed on (**center_x**,**center_y**).
CONFLICT ERROR: Operator **sign** with size **op_size** can not be placed on (**center_x**,**center_y**).

where **sign** is the sign of the operator, **center_x** and **center_y** are center location of the operator (defined in the superclass **Operator**), and **op_size** is the size of the operator (defined in the superclass **Operator**).

Instructions on The Placed Operators

The **move_operator(int x, int y, char direction, int move_by)** method moves the center of the operator by a specified amount to the given direction ('R' for right, 'L' for left, 'U' for up, 'D' for down). Here, **x** and **y** specifies a cell on the grid on which **a part of the operator** resides. The cell (**x**, **y**) **DON'T HAVE TO** be the center of the operator. You must first find the center, and move the operator by the amount **move_by** (e.g., see Figure 6) .

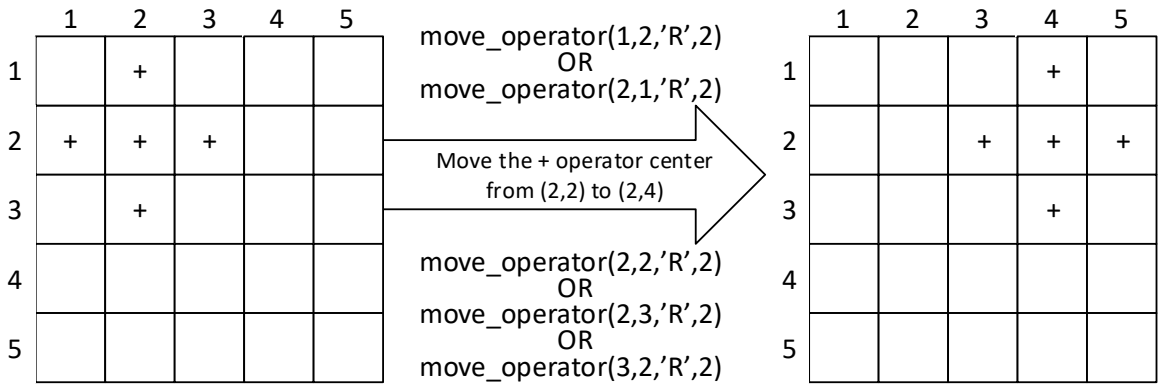

Figure 6: move_operator() Example For Right Direction

If the move doesn't result in any border and/or conflict error, then **move_operator()** method returns **true.** Otherwise, it returns **false**. In addition, at the end of the **move_operator()** method:

a) Print out following message if the operator is successfully moved:

SUCCESS: **sign** moved from (**x_before**,**y_before**) to (**x_after**,**y_after**).

where **sign** is the sign of the operator moved, **x_before** and **y_before** are center location of the operator before the move, and **x_after** and **y_after** are center location of the operator after the move.

b) If there is a **BORDER ERROR** (see Figure 4), then omit the move instruction and print out the following message:

BORDER ERROR: **sign** can not be moved from (**x_before**,**y_before**) to (**x_after**,**y_after**).

where **sign** is the sign of the operator that can't be moved, **x_before** and **y_before** are center location of the operator before the move attempt, and **x_after** and **y_after** are center location of the operator for the target move.

c) If there is a **CONFLICT ERROR** (see Figure 5), then omit the move instruction and print out the following message:

CONFLICT ERROR: **sign** can not be moved from (**x_before**,**y_before**) to (**x_after**,**y_after**).

where **sign** is the sign of the operator that can't be moved, **x_before** and **y_before** are center location of the operator before the move attempt, and **x_after** and **y_after** are center location of the operator for the target move.

d) If there is both **BORDER and CONFLICT ERRORs,** then omit the move instruction and print out the following messages **in order**:

BORDER ERROR: **sign** can not be moved from (**x_before**,**y_before**) to (**x_after**,**y_after**).
CONFLICT ERROR: **sign** can not be moved from (**x_before**,**y_before**) to (**x_after**,**y_after**).

where **sign** is the sign of the operator that can't be moved, **x_before** and **y_before** are center location of the operator before the move attempt, and **x_after** and **y_after** are center location of the operator for the target move.

print_operators() method calls **print_operator()** method on operators that are added to the **operators** instance variable. **Please don't forget that only successfully placed operators are added to the operators instance variable in their placement order (i.e., previously inserted operator has lower index in the operators array).** Thus, they must be printed out in the order they were inserted into the **operators** instance variable.

Submission and Rules

1. Your source code file must have the name **assignment2.cpp**.
2. Your program will be compiled using the following command on a Linux system. If it cannot be compiled and linked using this command, it will not be graded (failed submission).

   g++ -Wall -Werror assignment2.cpp -o assignment2

3. Your program will be checked using **Calico** (**https://bitbucket.org/uyar/calico)** automatic checker. Therefore, make sure you **print the messages exactly as given in the homework definition.**
4. You are **allowed to add** other helper methods to the given class definitions. However, beware that **main()** method of the **test files** will **only** use the given class definitions to construct instances of **Operator**(s), **ArithmeticOperator**(s), and **OperatorGrid**(s). The method calls on the created instances will also be done based on the above class definitions. **Your code will fail in tests if you don't obey the class definitions given above.**
5. Make sure your coding style is proper and consistent. Use the **clang-format tool** if necessary. Don't use any variable names in a language other than English.
6. Add comments to make your source code easy to understand.
7. This is an **individual** assignment. Collaboration in any form is NOT allowed. **No working together**, No sharing code in any form including showing code to your classmates to give them ideas.
8. All the code you submit must be your own. Don't copy/paste any piece of code from any resource including anything you've found on the Internet.
9. The assignments will be **checked for plagiarism** using both automated tools and manual inspection. Any assignment involving plagiarism and/or infringement of intellectual property **will not be graded and is subject to further disciplinary actions**.