VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PROGRAMMING FUNDAMENTALS - CO1027

ASSIGNMENT 2

THE GENERAL OFFENSIVE AND UPRISING SPRING 1975
Part 2: Ho Chi Minh Campaign

Version 1.3

Ho Chi Minh City, 02/2024

ASSIGNMENT SPECIFICATION

Version 1.3

1 Learning outcomes

After completing this assignment, students review and make good use of:

- Function and calling function
- File Input/Output
- Pointer and dynamic memory allocation
- Object-oriented programming
- Singly linked list

2 Introduction

This major assignment (BTL) is inspired by the 50th anniversary of the success of the Spring 1975 General Offensive and Uprising by our military and people, which led to the complete liberation of the South and the reunification of the country.

In Part 1, we reenacted the Central Highlands Campaign, the opening campaign of the Spring 1975 General Offensive and Uprising. Immediately afterward (at the end of March 1975), our military and people continued to achieve victory in the Hue-Da Nang liberation campaign, completely changing the course of the war.

Amid the rapid developments—where "one day was worth twenty years"—on April 14, 1975, the Politburo decided to launch the Saigon-Gia Dinh Liberation Campaign, officially named the Ho Chi Minh Campaign. This decision once again demonstrated the strategic vision of the Politburo, the Central Military Commission, and the General Command in leading, directing, and orchestrating the conclusion of the revolutionary war.

In this assignment, you are required to implement classes to simulate the progression of the campaign. The details of the classes that need to be implemented will be specified in the tasks below.

3 Classes in the Program

This assignment utilizes Object-Oriented Programming to reenact the Ho Chi Minh Campaign. The objects in this assignment are represented through classes and described as follows.

Note:

- Students must carefully read the requirements to identify the derivative classes (subclasses), as they will not be explicitly described.
- Students should propose additional attributes, methods, or other classes to support the implementation of the classes in this assignment.
- Within the scope of this assignment, when referring to distance, students should understand it as **Euclidean distance**.
- Within the scope of this assignment, when calculated values need to be rounded to an integer, students must **round up**.
- Within the scope of this assignment, Any value that exceeds the threshold (if a range is specified) should be rounded to the nearest threshold.

3.1 Military Units

Each military force participating in the war consists of different military units. There are two main types of military units: Combat Vehicles (Vehicle) and Infantry Forces (Infantry).

Requirement: Implement the abstract class **Unit** with the following specifications:

1. Protected attributes:
   - quantity: an integer representing the number of this type of military unit in the army.
   - weight: an integer representing the contribution weight to different military capability indices.
   - pos: of type Position, representing the position of this unit type on the battlefield.
2. Public constructor:
   The constructor (public) takes parameters `quantity`, `weight`, and `pos`, which correspond to the attributes of the same name. It initializes the attributes with the given parameter values.

```
Unit(int quantity, int weight, const Position pos)
```

3. Virtual destructor with public access.
4. Pure virtual method **getAttackScore**, which returns an integer representing the unit's contribution to a specific capability index (capability indices include **LF** and **EXP** of that army).

```
virtual int getAttackScore() = 0;
```

5. Method **getCurrentPosition**, which returns the current `Position` of the moving object.

```
Position getCurrentPosition() const;
```

6. Pure virtual method **str**, which returns a string representation of the object's information.

```
virtual string str() const = 0;
```

3.2 Vehicles

Each army possesses a number of different types of combat vehicles (Vehicle). A vehicle type is considered a military unit (Unit).

We define the enum **VehicleType** as follows:

```
enum VehicleType {TRUCK, MORTAR, ANTIAIRCRAFT, ARMOREDCAR, APC, ARTILLERY,
    TANK}
```

Where:

- TANK - Tanks (e.g., T-54, M41)
- ARTILLERY - Artillery (e.g., 105mm, 130mm)
- ARMOREDCAR - Armored cars (e.g., BTR-152, M113)
- APC - Armored personnel carriers (e.g., BTR-60, M113)
- TRUCK - Military trucks (e.g., ZIL-157, GMC CCKW)
- MORTAR - Mobile mortars (e.g., 82mm, 120mm)
- ANTIAIRCRAFT - Anti-aircraft weapons (e.g., ZSU-23-4, DShK)

Requirement: **Implement** the class **Vehicle** with the following specifications:

1. Private attributes:
   - vehicleType: of type `VehicleType`, representing the type of vehicle.
2. Public constructor, declared as shown below. In addition to the parameters already mentioned in **Unit**, it includes additional parameters specific to **Vehicle**, with names and meanings corresponding to the attributes described above.

```
Vehicle(int quantity, int weight, const Position pos, VehicleType
    vehicleType)
```

3. Method **getAttackScore** (public), which returns a value computed using the formula for the contribution score of a specific vehicle type:

$$score = \frac{typeValue * 304 + quantity * weight}{30}$$

Where:

- $typeValue$ is the corresponding enum value of the vehicle type in `VehicleType`.
- $quantity, weight$ are the respective values of `quantity` and `weight` for that vehicle.

4. Method **str** (public): Returns a string representing the vehicle information in the following format:

```
Vehicle[attr_name=<attr_value>]
```

Where `attr_name` and `attr_value` are the attribute names and values of the corresponding vehicle. The order of attributes to be printed follows this sequence: `vehicleType`, `quantity`, `weight`, `pos` and separated by comma (",").

3.3 Infantry Forces

Each military force consists of different types of infantry (Infantry) serving within the army. An infantry type is considered a military unit (Unit).

We define the enum **InfantryType** as follows:

```
enum InfantryType {SNIPER, ANTIAIRCRAFTSQUAD, MORTARSQUAD, ENGINEER,
    SPECIALFORCES, REGULARINFANTRY}
```

Where:

- SNIPER – Sniper
- ANTIAIRCRAFTSQUAD – Anti-aircraft squad
- MORTARSQUAD – Mortar squad
- ENGINEER – Combat engineer
- SPECIALFORCES – Special forces
- REGULARINFANTRY – Regular infantry

Requirement: **Implement** the class **Infantry** with the following specifications:

1. Private attributes:
   - infantryType: of type `InfantryType`, representing the type of infantry.
2. Public constructor, declared as shown below. In addition to the parameters already mentioned in **Unit**, it includes additional parameters specific to **Infantry**, with names and meanings corresponding to the attributes described above.

```
Infantry(int quantity, int weight, const Position pos, InfantryType
    infantryType)
```

3. Method **getAttackScore** (public), which returns a value computed using the formula for the contribution score of a specific infantry type:

$$score = typeValue * 56 + quantity * weight$$

   Where:
   - $typeValue$ is the corresponding enum value of the infantry type in `InfantryType`.
   - $quantity, weight$ are the respective values of `quantity` and `weight` for that infantry unit.

   Special conditions:
   - If the infantry type is **SPECIALFORCES** and the weight is a **perfect square number**, the unit is classified as commandos, and an additional 75 points are added to the contribution score.
   - We define: The **personal number** of a non-negative integer in the year Y is the single-digit number obtained by summing the digits of that number and the digits of Y, repeatedly summing the digits of the result until only a single-digit number remains.
     In this case, the combat year is 1975. After considering the commando case, if the **personal number** of the unit's score exceeds 7, the unit receives additional reinforcements of 20% of its current quantity. Conversely, if the personal number is less than 3, 10% of the current quantity will desert immediately. Adjust the *quantity* attribute accordingly and recalculate the *score*.
4. Method **str** (public): Returns a string representation of the infantry unit's information in the following format:

```
Infantry[attr_name=<attr_value>]
```

Where `attr_name` and `attr_value` are the attribute names and values of the corresponding infantry unit. The order of attributes to be printed follows this sequence: `infantryType`, `quantity`, `weight`, `pos` and separated by comma (",").

3.4 Army

Requirement: **Implement** the class **Army** with the following specifications:

1. Protected attributes:
   - **LF**: of type `int`, with a value in the range [0, 1000]. This value is calculated as the sum of the *score* from all combat vehicles (**Vehicle**) that this army possesses.
   - **EXP**: of type `int`, with a value in the range [0, 500]. This value is calculated as the sum of the *score* from all infantry forces (**Infantry**) that this army possesses.
   - **name**: of type `string`, representing the name of the army.
   - **unitList**: of type `UnitList*`, representing the list of military units this army possesses. Further details will be provided in later sections.
   - **battleField**: of type `BattleField*`, representing the battlefield where this army fights.
2. Public constructor that takes an array of military units, the size of this array, the army's name, and the battlefield where the army fights. The constructor's task is to compute the appropriate values for **LF** and **EXP**, and add the units to `unitList` accordingly.

```
Army(const Unit** unitArray, int size, string name, BattleField*
    battleField)
```

3. The fight method (**fight**) is a pure virtual function that takes an `Army*` (enemy) as the opponent of the current army and a boolean parameter `defense`, describing whether the battle is defensive or offensive. The default value is `false` (indicating an offensive battle). The return type is `void`.

```
virtual void fight(Army* enemy, bool defense = false) = 0;
```

4. Pure virtual method **str**, which returns a string representation of the object.

```
virtual string str() const = 0;
```

3.4.1 Liberation Army

In this campaign, our forces include the People's Army of Vietnam and the Liberation Army of South Vietnam (established in 1961). In this assignment, these are collectively referred to as the

Liberation Army. To represent the Liberation Army, we define a class named **LiberationArmy**.

Requirement: **Implement** the class **LiberationArmy** with the following specifications:

1. Constructor with the following signature:

```
LiberationArmy(const Unit** unitArray, int size, string name,
    BattleField* battleField)
```

2. Fight method (**fight**):

```
void fight(Army* enemy, bool defense = false)
```

- **Case: Attack (defense = false)**
  - Since the army has an offensive advantage, both **LF** and **EXP** are multiplied by 1.5.
  - Choosing a combined attack strategy: To optimize the attack plan, the army selects a combination of infantry units with the smallest total score greater than the enemy's **EXP**, referred to as **Combination A**. Additionally, it selects a combination of vehicles with the smallest total score greater than the enemy's **LF**, referred to as **Combination B**.

    If both combinations A and B are found, meaning a victory is achieved, remove these units from the unit list.

    If only one of the two combinations meets the criteria, the battle results in a stalemate. However, since the Liberation Army has an offensive advantage, if the overall combat index of the army in the unmatched category is higher than the enemy's corresponding index, victory is still secured. The scores are updated, the valid combination is removed, and all units in the unmatched combination are eliminated. If the index is lower, the battle does not take place.

    If neither combination is found, the battle does not occur.

    If the battle is won, all military units (including infantry forces) of the enemy will be confiscated and repurposed. These units will be added to the Liberation army's unit list (if adding is possible; else if they already exist, their quantity will be updated), and all indices (**LF** and **EXP**) will be recalculated.

    If no battle occurs, each unit in the army loses 10% of its weight. The army's combat indices should be updated accordingly.

    Hint: It is recommended to write helper methods and choose appropriate modifiers for them to maintain OOP principles. Additionally, since there are many cases where combat indices need to be updated, an efficient update mechanism should be implemented.

- **Case: Defense (defense = true)**
  - The Liberation Army is always combat-ready. Even when defending, it maintains an active stance, and all combat indices are multiplied by 1.3.
  - Compare the corresponding combat indices of both armies. If neither of the Liberation Army's indices is lower than the enemy's, the Liberation Army wins. If one of the two indices is lower, the army does not lose but will suffer a 10% reduction in quantity for each military unit. If both indices are lower, the Liberation Army will require reinforcements. Each military unit's quantity will be increased to the nearest **Fibonacci number**. After updating the indices, the combat situation should be re-evaluated accordingly.

3. Method **str**: Returns a string representation of the Liberation Army.

```
LiberationArmy[attr_name=<attr_value>]
```

The attributes should be printed in the following order: `name`, `LF`, `EXP`, `unitList`, and `battleField` and separated by comma (",").

3.4.2 ARVN (Army of the Republic of Vietnam)

At this time, the Saigon government's military no longer had direct American combat support. However, the U.S. still provided financial aid, weapons, logistics, and military advisory support.

Requirement: **Implement** the class **ARVN** (Army of the Republic of Vietnam) with the following specifications:

1. Constructor with the following signature:

```
ARVN(const Unit** unitArray, int size, string name, BattleField*
    battleField)
```

2. Fight method (**fight**):

```
void fight(Army* enemy, bool defense = false)
```

- **Case: Attack (defense = false)**
  - Although on the offensive, the ARVN suffered from internal divisions, low morale, and a lack of direct U.S. support. As a result, their combat indices remain unchanged in both offensive and defensive situations.

  - When the ARVN attacks, meaning the Liberation Army is defending, the Liberation Army will not be defeated even when defending. The ARVN will lose the battle and must reduce the quantity (*quantity*) of each military unit by 20%. The corresponding combat indices should be updated. If any military units have a quantity (*quantity*) of 1, remove them from the list and update the indices.

- **Case: Defense (defense = true)**
  - In this case, the Liberation Army is attacking. According to the previous descriptions, the Liberation Army may engage in battle and achieve victory. In this scenario, the ARVN's military units will be confiscated and repurposed (as described above), meaning these units should be removed from the ARVN's unit list. Since the ARVN has lost, the remaining military units will suffer a 20% reduction in weight (*weight*). If no battle occurs, the ARVN forces remain intact.

3. Method **str**: Returns a string representation of the ARVN forces.

   ```
   ARVN[attr_name=<attr_value>]
   ```

   The attributes should be printed in the following order: `name`, `LF`, `EXP`, `unitList`, and `battleField` and separated by comma (",").

3.5 Military Unit List

To manage a military force's units, we represent them as a **singly linked list**. The operations that can be performed on this list include:

- Adding a unit to the list (**insert** method): If the unit is a vehicle (**Vehicle**), it should be added to the end of the list. If the unit is an infantry unit, it should be added to the front. If the unit already exists in the list, its quantity (**quantity**) should be updated accordingly.
- Checking whether a military unit exists in the list (**isContain** method). This method takes a **VehicleType** or **InfantryType** as input and returns **True** if found; otherwise, it returns **False**.

A number is considered a **special number** if it can be expressed as the sum of distinct powers of a given base k. For example, ***10 is a special number in base 3*** because $10 = 3^2 + 3^0$.

Each army has a maximum number of units it can store in the list. This number depends on the total strength $S = \textbf{LF} + \textbf{EXP}$ of the army:

- If S is a **special number** in at least one of the **odd prime bases less than 10**, then the maximum list capacity is 12.
- In all other cases, the capacity is 8.

Requirement: **Implement** the class **UnitList** with the following specifications:

1. Attributes: Students should define the necessary attributes, with one mandatory attribute being **capacity** (of type **int**, private). Implement a corresponding **constructor** to initialize this in the army classes. Ensure that:
   - If the unit is a vehicle (**Vehicle**), it should be added to the end of the list.
   - If the unit is an infantry unit, it should be added to the front of the list.
2. Methods as described above, including:

```
bool insert (Unit* unit); // Returns true if insertion is successful
bool isContain(VehicleType vehicleType); // Returns true if it exists
bool isContain(InfantryType infantryType); // Returns true if it exists
string str() const;
```

   For the **str()** method, the string format should be:

```
UnitList[count_vehicle=<v_c>;count_infantry=<i_c>;<unit_list>]
```

   Where:

   - `<v_c>`, `<i_c>`: Represent the number of vehicle units and infantry units in the list, respectively.
   - `<unit_list>`: A string representation of the units from the start to the end of the linked list. Each unit should be printed in the format of its corresponding **Unit str()** method, separated by commas (,).

3.6 Position

The class **Position** represents a location in the program. **Position** can be used to store the positions of military units.

1. Two private attributes, **r** and **c**, both of type `int`, representing the row and column positions, respectively.

2. Public constructor with two parameters, **r** and **c**, which are assigned to the row and column attributes. These parameters have default values of 0.

```
Position(int r=0, int c=0);
```

3. Public constructor with one parameter, **str_pos**, which represents a position in string format. The format of **str_pos** is `"(<r>,<c>)"`, where `<r>` and `<c>` are the respective row and column values.

```
Position(const string & str_pos); // Example: str_pos = "(1,15)"
```

4. Four getter and setter methods for the row and column attributes, declared as follows:

```
int getRow() const ;
int getCol() const ;
void setRow(int r) ;
void setCol(int c) ;
```

5. The **str** method returns a string representation of the position. The return format matches the format of **str_pos** in the constructor. For example, if $r = 1, c = 15$, then **str** returns **"(1,15)"**.

```
string str() const;
```

6. Additional methods may be required in subsequent sections.

3.7 Terrain Elements of the Battlefield

Each battlefield has terrain elements that affect the combat capabilities of the armies.

The terrain elements include:

- Road (ROAD)
- Forested Mountains (FOREST)
- Deep River (RIVER)
- Fortification (FORTIFICATION)
- Urban Area (URBAN)
- Demilitarized Zone (temples, churches, hospitals, etc.) (SPECIAL ZONE)

Each terrain element impacts the combat indices of the armies differently.

Requirement: Implement the class **TerrainElement** with the following specifications:

1. Constructor and destructor of the class.

2. Pure virtual method **getEffect**, which determines the effect of the terrain element on the combat indices of the army.

```
virtual void getEffect(Army* army) = 0;
```

Requirement: Implement the classes **Road, Mountain, River, Urban, Fortification, and SpecialZone** to concretely define the terrain elements. Each of these classes should implement the **getEffect** method according to the specific descriptions outlined in the following table:

Bảng 1: Effects of Terrain on Military Units

No.	Terrain	Effect on Liberation Army	Effect on ARVN
1	Road	No effect	No effect
2	Forested Mountains	While the infantry forces are highly proficient in combat in mountainous terrain, military vehicles are not suited for movement in these areas. Within a radius of 2 units from the mountain terrain position: • **EXP** increases by 30% of each *attackScore* for infantry forces within the affected radius. • **LF** decreases by 10% of each *attackScore* for combat vehicles within the affected radius.	Since this is a base area for ARVN, the influence range is wider, although the effect is lower. Within a radius of 4 units from the mountain terrain position: • **EXP** increases by 20% of each *attackScore* for infantry forces within the affected radius. • **LF** decreases by 5% of each *attackScore* for combat vehicles within the affected radius.

No.	Terrain	Effect on Liberation Army	Effect on ARVN
3	Deep River	Infantry movement speed is restricted. *attackScore* of infantry units within a radius of 2 units is reduced by 10%.	*attackScore* of infantry units within a radius of 2 units is reduced by 10%.
4	Urban Area	• If the infantry unit is either **SPECIAL-FORCES** or **REGULARINFANTRY**, and is within a radius of 5 units, each *attackScore* of these infantry units is increased by: $\frac{2 * attackScore}{D}$ where D is the distance between the urban area and the corresponding infantry unit. Other infantry forces are not affected. • Combat vehicles of type **ARTILLERY** will have their *attackScore* reduced by 50% if within a radius of 2 units.	• If **REGULARINFANTRY** units are within a radius of 3 units, each *attackScore* is increased by: $\frac{3 * attackScore}{2 * D}$ where D is the distance between the urban area and the corresponding infantry unit. Other infantry forces are not affected. • ARVN's combat vehicles are not affected in urban areas.

No.	Terrain	Effect on Liberation Army	Effect on ARVN
5	Fortification	Creates difficulties for the Liberation Army in advancing. All military units within a radius of 2 units from the Liberation Army are reduced by 20% in *attackScore*.	ARVN has a strategic advantage in these locations, gaining a 20% increase in *attackScore* for each military unit within a radius of 2 units.
6	Demilitarized Zone	No battles may occur. All military units within a radius of 1 unit around this area will have their *attackScore* reduced to 0.	

3.8 Battlefield

The Ho Chi Minh Campaign took place across five battlefronts, each with different battles. Therefore, the battlefield for each front should be defined as a class. Assume that each battlefield is represented as a 2D array.

Requirement: Implement the class **BattleField** with the following specifications:

1. Private attributes:
   - **n_rows** and **n_cols** (of type `int`), representing the number of rows and columns of the battlefield map.
   - **terrain**, a private attribute representing a 2D array where each element of the array is a `TerrainElement`. Students should choose an appropriate data type for the **terrain** attribute.
2. Method **str**: Returns the string information of the battlefield

   ```
   BattleField[attr_name=<attr_value>]
   ␣␣␣␣␣
   ```

   The attributes are ordered sequentially as n_rows, n_cols and separated by a comma (",").
3. Constructor with the following signature:

```
1 BattleField(int n_rows, int n_cols, vector<Position*> arrayForest,
```

```
2           vector<Position*> arrayRiver, vector<Position*>
    arrayFortification,
3           vector<Position*> arrayUrban, vector<Position*>
    arraySpecialZone)
```

Where:

- `n_rows`, `n_cols`: The number of rows and columns of the battlefield map.
- Arrays with the naming format **array**<**Element**> represent the positions of terrain elements on the map. Any positions not included in these arrays are considered roads (ROAD).

4. Public destructor to release dynamically allocated memory.
5. Additional methods may be required in later sections.

3.9 Configuration

A configuration file is used to define the settings for the program. The file consists of lines, each following one of the formats below. Note that the order of the lines may vary.

1. `NUM_ROWS=<nr>`
2. `NUM_COLS=<nc>`
3. `ARRAY_FOREST=[<pos1>,<pos2>,..]`
4. `ARRAY_RIVER=[<pos1>,<pos2>,..]`
5. `ARRAY_FORTIFICATION=[<pos1>,<pos2>,..]`
6. `ARRAY_URBAN=[<pos1>,<pos2>,..]`
7. `ARRAY_SPECIAL_ZONE=[<pos1>,<pos2>,..]`
8. `UNIT_LIST=[<unit1>,<unit2>,...]`
9. `EVENT_CODE=<ec>`

Where:

1. `<nr>` represents the number of rows in the battlefield map, for example:
   `NUM_ROWS=10`
2. `<nc>` represents the number of columns in the battlefield map, for example:
   `NUM_COLS=10`
3. `[<pos1>,<pos2>,..]` represents the positions of terrain elements on the battlefield. Example:
   `ARRAY_FOREST=[(1,2),(2,3),(3,4)]`
   represents the positions of forest elements (FOREST) on the battlefield.

4. `[<unit1>,<unit2>,...]` represents the list of military units for both armies. Each unit follows the format:
   `UNIT_NAME(quantity,weight,position,armyBelong)`
   where `armyBelong=0` corresponds to the Liberation Army, and `armyBelong=1` corresponds to ARVN.
   Example:

```
UNIT_LIST=[
TANK(5,2,(1,2),0),
TANK(5,2,(3,2),1),
REGULARINFANTRY(5,2,(1,1),0),
REGULARINFANTRY(5,2,(3,3),1)
]

```

5. `<ec>` represents the event code used to determine which army is on the offensive or defensive in the current battle.

Requirement: Implement the class **Configuration** to represent a program configuration by reading the configuration file. The class **Configuration** has the following specifications:

1. Private attributes:
   - **num_rows, num_cols**: Represent the number of rows and columns of the battlefield map.
   - **arrayForest, arrayRiver, arrayFortification, arrayUrban, arraySpecialZone**: of type *vector<Position*>*, representing the positions of terrain elements on the battlefield.
   - **liberationUnits**: of type *Unit**, representing an array of military units belonging to the Liberation Army.
   - **ARVNUnits**: of type *Unit**, representing an array of military units belonging to ARVN.
   - **eventCode**: of type *int*, representing the event code that determines which army is in the offensive position. The event code is in the range [00,99]. If the input value exceeds 99, the last two digits of the number will be used as the event code. If the input value is below 00, the event code will default to 00.
2. Constructor **Configuration** is declared as follows. The constructor takes **filepath**, a string representing the path to the configuration file, and initializes the attributes according to the specifications above.

```
Configuration(const string & filepath);
```

3. The destructor must free all dynamically allocated memory.
4. The method **str** returns a string representation of the Configuration.

```
string str() const;
```

The format of this string is:

```
Configuration[
<attribute_name1>=<attribute_value1>...
]
```

Each attribute and its corresponding value will be printed in the order listed in the "Private attributes" section of this class. For each military unit, the string representation will be obtained by calling the corresponding **str** method of the unit.

3.10 Class HCMCampaign

The battle on a battlefield is conducted by calling the fight method of the corresponding armies. After a battle between two military units, if a unit's $attackScore \leq 5$, it will be removed from its army's military unit list.

Specific rules:

- If the event code is **less than 75**: The Liberation Army is on the offensive, while ARVN is defending.
- If the event code is **75 or greater**: ARVN initiates the attack. Once ARVN's attack concludes, the Liberation Army will immediately counterattack.

Requirement: Students should propose and implement the **HCMCampaign** class according to the following requirements:

1. Attributes:
   - **config** (Type: `Configuration*`)
   - **battleField** (Type: `BattleField*`)
   - **liberationArmy** (Type: `LiberationArmy*`)
   - **ARVN** (Type: `ARVN*`)
2. Constructor that takes one parameter: the file path to the program's configuration file. The constructor initializes all necessary class information.

```
HCMCampaign(const string & config_file_path)
```

3. Method **run**: No input parameters. This method executes the battle between the two armies on the specified battlefield according to the described sequence. At the start of the battle, terrain elements affect combat indices before engagement begins.
4. Method **printResult** (Return type: `string`): No input parameters. Outputs the combat indices of both armies after the battle in the following format:

   ```
   LIBERATIONARMY[LF=<LF>,EXP=<EXP>]-ARVN[LF=<LF>,EXP=<EXP>]
   ```

   where `<LF>` and `<EXP>` represent the **LF** and **EXP** values of the corresponding army.

4 Requirements

To complete this major assignment, students must:

1. Read the entire description file.
2. Download the `initial.zip` file and extract it. After extraction, students will receive the files: `main.cpp`, `main.h`, `hcmcampaign.h`, `hcmcampaign.cpp`, and sample data files. Students must submit only `hcmcampaign.h` and `hcmcampaign.cpp` and must not modify `main.h` when testing the program.
3. Students should use the following command to compile the program:
   g++ -o main main.cpp hcmcampaign.cpp -I . -std=c++11
   To run the program, use the command:
   ./main
   These commands should be used in the command prompt/terminal for compilation and execution. If students use an IDE, they must ensure:
   - All necessary files are added to the project/workspace.
   - The compilation command includes `hcmcampaign.cpp`, `-std=c++11`, and `-I .`.
   - IDEs typically provide **Build** and **Run** buttons. When pressing **Build**, the IDE executes a corresponding compilation command, often compiling only `main.cpp`. Students must configure the IDE to modify the compilation command to include the necessary files and options.
4. The program will be graded on a Unix-based platform. The grading platform and the student's compiler may differ. The LMS submission system is configured to match the

grading environment. Students must test their program in the LMS submission system and resolve any errors there to ensure correct results in the final grading.

5. Modify `hcmcampaign.h` and `hcmcampaign.cpp` to complete this assignment while ensuring the following:
   - The only `#include` directive allowed in `hcmcampaign.h` is: **#include "main.h"** and in `hcmcampaign.cpp`: **#include "hcmcampaign.h"**. No other `#include` statements are allowed in these files.
   - Implement all functions as described in the tasks within this assignment.
   - Ensure all classes follow the fundamental principles of Object-Oriented Programming.
6. Students are encouraged to create additional classes and methods to complete the assignment.
7. Students must implement all classes and methods listed in this assignment. Failure to do so will result in a score of 0. If a method or class cannot be fully implemented, students must at least provide an empty implementation using curly braces {}.

5 Submission

Students must submit only two files: `hcmcampaign.h` and `hcmcampaign.cpp` before the deadline specified in "Assignment 2 - Submission". Some basic test cases will be used to check whether the submission compiles and runs correctly. Students can submit multiple times, but only the last submission will be graded.

Due to system constraints, a high volume of submissions near the deadline may cause technical issues. Students should submit their work as early as possible. Submissions made in the last hour before the deadline are at the student's own risk. Once the deadline passes, the system will close, and no further submissions will be accepted. Submissions through any other method will not be considered.

6 Additional Rules

- Students must complete this assignment independently and prevent others from copying their work. Any violations will be subject to the university's academic integrity policies.
- All decisions made by the assignment instructor are final.

- Students will not receive test cases after grading. However, the assignment score breakdown will be provided.
- The content of this assignment is harmonized with a question in the final exam that covers a similar topic.

7 Harmony for the Major Assignment

The final exam will include questions related to the content of this assignment.

Students must complete the assignment using their own knowledge. If a student cheats on this assignment, they will be unable to answer the harmony question in the final exam and will receive a score of 0 for this assignment.

Students **must** complete the harmony question in the final exam. Failure to do so will result in a score of 0 for this assignment and failure in the course. **No exceptions or explanations will be accepted.**

8 Academic Integrity and Cheating

The assignment must be completed **individually**. A student will be considered to have cheated if:

- There is an abnormal similarity between submitted code. In such cases, **all submissions involved** will be treated as cheating. Students must protect their source code.
- A student's code is submitted by another student.
- The student does not understand their own submitted code, except for the provided starter code. Students may refer to any resources but must fully understand the meaning of every line they write. If a student does not understand the code they are referencing, they are strongly warned **not to use it**; instead, they should apply concepts learned in class.
- The student accidentally submits another student's work under their own account.
- The student uses source code generated by automated tools without understanding it.

If a student is found guilty of cheating, they will receive a **zero for the entire course** (not just the assignment).

NO EXPLANATIONS WILL BE ACCEPTED, AND NO EXCEPTIONS WILL BE MADE!

After submission, some students may be randomly selected for an interview to verify that their work was completed independently.

9 Changes from Previous Versions

- Add a requirement to round to the threshold when a value exceeds it.
- For the str methods: include a description of the attributes separated by commas.
- Add a str method for the class BattleField.
- Modify the capacity specification for the class UnitList.
- Adjust the description of function **getEffect** of terrain type RIVER
- Modify the *attackScore* formula of vehicle
- Modifiy the description of class **HCMCapaign**

10 Conclusion

The Spring 1975 General Offensive and Uprising was a resounding success. On April 30, 1975, the Saigon government declared unconditional surrender, leading to the liberation of the South and the complete reunification of the country. This marked the beginning of a new era for the nation—an era of independence, peace, self-reliance, and strength.

In celebration of the 50th anniversary of the liberation of South Vietnam and national reunification (April 30, 1975 - April 30, 2025), we hope that through this assignment, students will revisit the nation's heroic history and cultivate a spirit of patriotism to contribute to the country's continued growth and development as it advances into a new era.

GOOD LUCK WITH YOUR ASSIGNMENT!