COMP 2404 B/C – Assignment #5

Due: Thursday, April 4 at 11:59 pm

1. Goal

For this assignment, you will write a program in C++, in the Ubuntu Linux environment of the course VM, that simulates the escape of two heroes, Timmy Tortoise and Prince Harold the Hare, out of a deep Pit full of snake orcs (snorcs). Fortunately for our heroes, their good friend Lady Gwendolyn anticipated their capture, so she sent a group of highly trained stealth Ninjas to rescue our heroes. But in a tragic and unforeseen turn of events, the Ninjas are highly vulnerable to snorc bites, and they turn into lethal giant snorcs once bitten.

Your program will simulate the attempted escape by our heroes and the rescue attempts by the Ninjas, as well as the efforts of the snorcs and giant snorcs to stop them. The movements of each participant must be randomly determined, and your code must display the simulation as it progresses. It must print out the outcome at the end, specifically whether the heroes escaped from the Snorc Pit or died in the attempt.

NOTE: This assignment is built along the same theme as Assignment #4. Assignment #5 *can be implemented as a stand-alone*, so you **DO NOT** need to have completed Assignment #4 in order to do Assignment #5. However, there are bonus marks available for the full integration of the code for both assignments.

2. Learning Outcomes

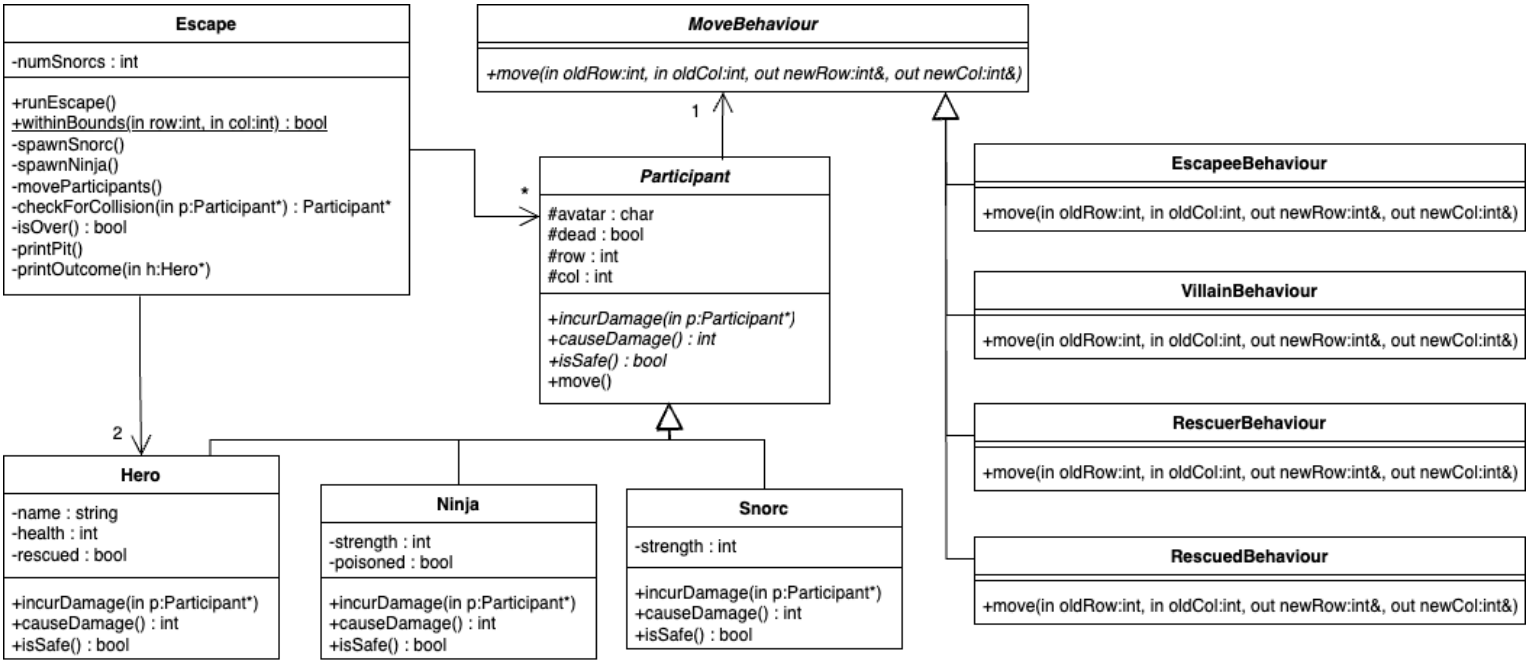
With this assignment, you will:

- implement a simplified Strategy design pattern using polymorphism and dynamic binding
- practice implementing a collection class template, and some overloaded operators
- optionally, fully integrate this program with your code for Assignment #4

3. Instructions

3.1. Understand the UML class diagram

You will begin by understanding the UML class diagram below. Your program will implement the objects and class associations represented in the diagram, as they are shown. UML class diagrams are explained in the course material in section 2.3.



3.2. **Understand the provided base code**

The `a5-posted.tar` file contains the required constant definitions that your program must use. A global `random()` function has also been provided, and you must use it throughout the program to randomize the participant behaviour. The pseudorandom number generator (PRNG) must be seeded **once** at the beginning of the program, using the statement: `srand( (unsigned)time( NULL ) );`

3.3. **Understand the overall rules of the escape**

3.3.1. The escape simulates the behaviour of multiple participants: Timmy and Harold are trying to climb up the slippery walls and out of the Snorc Pit, and the snorcs are trying to stop them. Ninjas are rappelling down the Pit to rescue our heroes, but if a Ninja is bitten by a snorc, they turn into a giant snorc that is even more lethal to our heroes.

3.3.2. The Snorc Pit is displayed as a 2D grid of rows and columns, with the origin point `(0,0)` located in the top-left corner.

3.3.3. Timmy and Harold begin their escape on the bottom row of the Snorc Pit, at a randomly determined column between 7 and 16 inclusively, and they *cannot* begin at the same column as each other. A hero successfully escapes the Pit if they reach the top row without dying. Each hero begins the escape with a health indicator at 20 points, and they lose health points every time they are bitten by a snorc, which happens during a collision. If a hero's health indicator reaches zero, that hero dies.

3.3.4. Snorcs are randomly spawned in the bottom rows of the Pit, up to a maximum of 12 snorcs in total, and they cannot climb higher than the bottom seven rows.

3.3.5. Ninjas are randomly spawned at the top of the Pit, and they rappel down the Pit in a straight line. If they collide with a hero, that hero is rescued, and both hero and Ninja are pulled up and out of the Pit in a straight line. If a Ninja collides with any kind of snorc, they become a giant snorc, and they start behaving exactly like the regular snorcs, but deadlier, as they try to bite and stop our heroes.

3.3.6. Every participant is represented by an avatar, which is defined as a single character. Timmy's avatar is `'T'`, Harold's is `'H'`, and every Ninja's is `'N'`. A snorc is represented as `'s'` (lowercase `s`), and a giant snorc is represented as `'S'` (uppercase `S`). When a hero dies, their avatar is changed to a cross `'+'` that represents their grave.

3.3.7. Your program must declare a single instance of the escape, stored as an `Escape` object, declared locally in the `main()` function, which calls its `runEscape()` member function. The `Escape` object then takes over the control duties of the program.

3.3.8. All participants in the escape, including heroes, Ninjas, regular snorcs and giant snorcs, must be *dynamically allocated* and stored as pointers in **the same participants collection** together. This is mandatory so that polymorphism can be implemented. **Do not** assume a specific order for participants in the collection.

3.3.9. The escape is implemented as a continuous escape loop that executes until each hero has either escaped the Pit by reaching the top, or died in the attempt.

3.3.10. The simulation begins with no snorcs in the Pit. At every iteration of the escape loop, there is a 90% probability that a new snorc is spawned and joins the others in attempting to stop our heroes. A maximum of 12 snorcs can be spawned in total. A newly spawned snorc is initially positioned in a randomly determined row in the bottom 5 rows inclusively, and a randomly generated valid column. Each snorc is spawned with a randomized amount of strength, between 2 and 4 inclusively.

3.3.11. The simulation begins with no Ninjas. At every iteration of the escape loop, there is a 33% probability that a new Ninja is spawned and joins the others in attempting to rescue our heroes. A newly spawned Ninja is initially positioned at the top of the Pit and a randomly generated valid column. As long as the Ninja has not been poisoned by getting bitten by a snorc, their strength remains at zero.

3.3.12. At every iteration of the escape loop, for every participant (hero, Ninja, snorc, giant snorc):

(a) a new position is computed **polymorphically** for that participant, based on random values, as described in instruction 3.4, and the participant's position is updated to the newly computed one

(b) once a participant is moved to its new position, the entire participants collection must be traversed and every *other* participant's position must be compared with the newly computed one, to determine if a collision has occurred (both participants occupy the exact same row and column)

(c) if a collision has occurred, it must be handled **polymorphically**; for most collisions, nothing happens; if a hero and a snorc (regular or giant) collide, the snorc bites the hero, and the hero's

health is decreased by the amount of strength possessed by that snorc; if a hero and a Ninja collide, the Ninja rescues the hero, and both begin climbing up the Snorc Pit in a straight line; if a snorc (regular or giant) and a Ninja collide, the Ninja is poisoned and becomes a giant snorc, with a randomly generated amount of strength greater than a regular snorc

3.3.13. At the end of every iteration of the escape loop, the Pit must be printed out, as well as both heroes' names, health points, and current status (Escaped, Rescued, Deceased, or nothing if they are still escaping), next to the bottom two rows of the Pit, as seen in the workshop video.

3.3.14. Once the escape loop has concluded, the Snorc Pit and both hero names, health points, and status must be printed to the screen one final time, and the outcome of the escape must be printed out. For each hero, the program must state that either they escaped on their own, or they were rescued, or they died.

3.4. **Understand the move behaviours**

Every participant will have a move behaviour object that determines how they move from one position to another in the Snorc Pit, in accordance with the Strategy design pattern that we covered in the in-class coding examples of section 3.3, program #8. In addition to an abstract class used as the base class, the following types of behaviour classes must be implemented:

3.4.1. *Escapee behaviour*:

(a) this is the behaviour of each hero before they are rescued by a Ninja

(b) the hero moves from their current row by a randomly determined one or two rows up, or one or two rows down, or they stay in the same row

(c) they move from their current column by a randomly determined one column to the left, or one column to the right, or they stay in the same column

3.4.2. *Villain behaviour*:

(a) this is the behaviour of all snorcs (regular and giant)

(b) the snorc moves from its current row by a randomly determined one row up, or one row down

(c) *snorcs cannot climb very high!* if the new row places the snorc higher than the bottom seven (7) rows of the Snorc Pit, then the snorc stays in its current row

(d) the snorc moves from its current column by a randomly determined one column to the left, or one column to the right, or stays in the same column

3.4.3. *Rescuer behaviour*:

(a) this is the behaviour of each Ninja before they rescue a hero

(b) the Ninja moves down one row from their current one

(c) they stay in the same column

3.4.4. *Rescued behaviour*:

(a) this is the behaviour of a rescued hero and a Ninja after it rescues a hero

(b) the hero or Ninja moves up one row from their current one

(c) they stay in the same column

3.4.5. Both heroes begin with escapee behaviour. If a hero is rescued by a Ninja, they both change to rescued behaviour.

3.4.6. A Ninja begins with rescuer behaviour when they are spawned. If they collide with a hero and rescue them, they change to rescued behaviour. If they collide with a snorc (regular or giant), they become a giant snorc and change to villain behaviour.

3.4.7. A regular snorc has villain behaviour from the time they are spawned, until the end of the simulation.

3.4.8. A giant snorc has villain behaviour from the time they are turned from a Ninja, until the end of the simulation.

3.5. **Implement the `Participant` class**

The `Participant` class is the base class for every kind of participant in the simulation. This class must be *abstract*, and its three derived classes, `Hero`, `Snorc`, and `Ninja`, must be *concrete*.

The `Participant` class contains the data members and member functions indicated in the given UML class diagram. In addition:

3.5.1. The default constructor must take four parameters: a participant's avatar, its initial row and column, and its move behaviour. All data members must be initialized accordingly.

3.5.2. The destructor must be virtual, and it must deallocate the required memory.

3.5.3. The `incurDamage()` member function must be **pure virtual**. It will be used by the simulation to determine how this participant reacts to the damage caused by another participant during a collision.

3.5.4. The `causeDamage()` member function must be **pure virtual**. It will be used by the simulation to determine how much damage is caused by this participant in a collision with another participant.

3.5.5. The `isSafe()` member function must be **pure virtual**. It will be used by the simulation to determine if a participant has left the Snorc Pit.

3.5.6. The `move()` member function delegates to the move behaviour object the work of computing a new row and column for the participant, provided that they are alive and still inside the Snorc Pit. All computed rows and columns must be valid within the Snorc Pit.

3.5.7. You will need getter member functions for the avatar, row, and column, as well as a setter member function for the move behaviour. You will implement a `bool isDead()` member function that indicates whether or not the participant is dead.

NOTE: DO NOT provide an implementation for any of the pure virtual functions in this class!

3.6. Modify the `List` class

You will modify the `List` class that we implemented in the coding example of section 3.1, program #8, so that it becomes a *class template*. You will also make the following changes:

3.6.1. You must remove the `add()`, `del()`, and `print()` member functions.

3.6.2. Each node must store data of the template type `T`. We can assume that this data will always be a pointer to a dynamically allocated object.

3.6.3. Implement the overloaded addition-assignment (`+=`) operator that takes a `T` parameter and adds that data to the back of the list. You must enable cascading. Everywhere that elements are added to the list, you must call the `+=` operator using the *short form*, for example `partsList += timmy;` **DO NOT** use the long form equivalent (for example `partsList.operator+=(timmy);`)

3.6.4. Implement `void convertToArray(T* arr, int& size)` member function that loops through the list and populates the parameter array with each data element in the list. The `size` parameter must be used to return the number of elements in the array (and in the list).

3.7. Implement the `Hero` class

The `Hero` class is derived from the `Participant` class and represents one of the two heroes (Timmy or Harold). It contains all the class members indicated in the UML class diagram. In addition:

3.7.1. The default constructor must take four parameters: the hero's avatar, initial row and column, and name. It must initialize all data members accordingly, and it must use base class initializer syntax to call the base class constructor. A hero's initial behaviour is escapee behaviour.

3.7.2. The virtual `incurDamage()` member function carries out the effects of a collision with the parameter participant. It must do the following:

(a) using **dynamic binding**, this function makes a *polymorphic* call to the parameter participant's `causeDamage()` member function to determine how much damage that other participant can inflict; **DO NOT** check the other participant's class type for any reason! you **must** use polymorphism instead

(b) if the damage amount corresponds to the rescue constant provided in the base code, it means that the other participant is a Ninja who is rescuing our hero; the hero's rescued data member must be set to true, and their behaviour must be changed to rescued behaviour

(c) otherwise, the hero's health must be reduced by the amount of damage caused by the parameter participant; if the hero's health reaches zero or less, the hero dies, and its avatar is changed to a `'+'` character to indicate the hero's grave

3.7.3. The virtual `causeDamage()` member function must always returns zero, as heroes do not cause any damage to other participants.

3.7.4. The virtual `isSafe()` member function returns true if the hero already reached the top of the Pit.

3.7.5. You will need getter member functions for the hero's name, health indicator, and rescued flag.

3.8. Implement the `Snorc` class

The `Snorc` class is derived from the `Participant` class and represents a regular snorc. It contains all the class members indicated in the UML class diagram. In addition:

3.8.1. The default constructor must take three parameters: the snorc's initial row and column, and strength. It must initialize all data members and call the base class constructor using correct syntax. A snorc's behaviour is always villain behaviour.

3.8.2. The virtual `incurDamage()` member function does nothing. Snorcs do not suffer any damage in a collision with another participant.

3.8.3. The virtual `causeDamage()` member function returns the snorc's strength, as this represents the amount of damage that they can cause to other participants.

3.8.4. The virtual `isSafe()` member function always returns false, since a snorc never leaves the Pit.

3.9. Implement the `Ninja` class

The `Ninja` class is derived from the `Participant` class and represents a Ninja. It contains all the class members indicated in the UML class diagram. In addition:

3.9.1. The default constructor must take two parameters: the Ninja's initial row and column. It must initialize all data members and call the base class constructor using correct syntax. A Ninja's initial behaviour is rescuer behaviour, and its strength is set to zero.

3.9.2. The virtual `incurDamage()` member function carries out the effects of a collision with the parameter participant. It must do the following:

(a) if the Ninja has been poisoned, the function does nothing; it means that the Ninja is now a giant snorc and does not suffer any damage in a collision with another participant

(b) otherwise, using **dynamic binding**, this function makes a *polymorphic* call to the parameter participant's `causeDamage()` member function to determine how much damage that other participant can inflict; **DO NOT** check the other participant's class type for any reason! you **must** use polymorphism instead

(c) if the damage amount corresponds to the rescue constant provided in the base code, it means that the other participant is a Ninja, so the function does nothing

(d) if the damage amount is zero, it means that the other participant is a hero, so the Ninja must now rescue the hero; the Ninja's behaviour is changed to rescued behaviour

(e) if the damage is any other amount, it means that the other participant is a snorc and the Ninja has now been poisoned and becomes a giant snorc; the Ninja's avatar is changed to a `'S'` character to indicate a giant snorc, its strength must be set to a random amount between 6 and 8 inclusively, its poisoned flag must be set to true, and its behaviour must change to villain behaviour

3.9.3. The virtual `causeDamage()` member function does the following: if the Ninja has been poisoned and is now a giant snorc, the function returns the strength data member as the amount of damage that this giant snorc can cause to other participants; otherwise, the function returns the rescue constant provided in the base code.

3.9.4. The virtual `isSafe()` member function returns true if the Ninja has left the Snorc Pit, either at the top where they helped a hero escape, or at the bottom where they reached the ground and disappeared. Giant snorcs never leave the Pit, so if a Ninja has been poisoned, they can never be safely out.

3.10. Implement the `MoveBehaviour` classes

You will implement an abstract `MoveBehaviour` class, as well as its four derived classes, in accordance with the given UML class diagram. The `move()` member function is pure virtual in the base class, and it must be implemented in each derived class, exactly as described in instruction 3.4.

3.11. **Implement the `Escape` class**

The `Escape` class serves as the control object for the simulation. It contains all the data members and member functions indicated in the given UML class diagram. In addition:

3.11.1. The escape's collection of participants must be stored as a statically allocated `List` object, using the class template defined in instruction 3.6. The data stored in the `List` must be **`Participant` pointers**. All participants in the escape **must** be stored in this same collection, otherwise polymorphism will not work.

3.11.2. The two heroes are stored in the escape's participants collection, but the escape also stores two separate pointers to the *same* `Hero` objects. The *only use* for these separate pointers must be to determine if the escape is over, and to print the Pit and eventually the outcome of the simulation.

3.11.3. The constructor must do the following:

(a) seed the PRNG, as described in instruction 3.2, and set the initial number of snorcs in the escape

(b) dynamically allocate both heroes with the correct values, including randomly generated initial columns in accordance with the escape rules, and add both heroes to the participants collection

3.11.4. The `runEscape()` member function implements the escape loop, exactly as described in the escape rules in instruction 3.3, and then it must print the outcome of the simulation for both heroes.

3.11.5. The `isOver()` member function determines if each hero has either escaped the Pit or died.

3.11.6. The static `withinBounds()` member function determines whether or not the given row and column represent a valid position within the boundaries of the Snorc Pit.

3.11.7. The `spawnSnorc()` member function dynamically allocates a new snorc with the correct randomly generated values, and adds it to the participants collection. The total number of snorcs must be updated to ensure that the maximum number is not exceeded.

3.11.8. The `spawnNinja()` member function dynamically allocates a new Ninja with the correct randomly generated values, and adds it to the participants collection.

3.11.9. The `checkForCollision()` member function converts the participants list to an array, and loops over the array to find a participant that is occupying the same position as the parameter participant. If one is found, the function returns that participant using the return value, otherwise it returns null.

3.11.10. The `moveParticipants()` member function converts the participants list to an array and loops over the array; for each participant, it does the following:

(a) using **dynamic binding**, make a *polymorphic* call to move the participant to a new row and column in the Snorc Pit

(b) check if the participant has collided with another participant

(c) if there is a collision and the other participant is neither dead nor safely out of the Snorc Pit, the two participants **both** *polymorphically* incur damage from each other

NOTE: DO NOT check any participant's class type anywhere! You must use polymorphism instead.

3.11.11. The `printPit()` member function must do the following:

(a) declare a temporary 2D grid of characters to represent the Snorc Pit and its participants, and initialize it with spaces

(b) declare a primitive array of `Participant` pointers, and convert the participants list into that array

(c) loop through the participants array, and place each participant's avatar in their current row and column in the temporary grid

(d) print the temporary grid to the screen

(e) on the bottom two lines, the names, health indicators, and current status of each hero must be printed out; if the hero is still alive and in the Snorc Pit, no status is printed, otherwise the status must indicate if the hero is escaped, rescued, or deceased

NOTE: The grid must be printed in the **exact same format** shown in the workshop video.

3.11.12. The `printOutcome()` member function indicates the outcome for the given hero, including whether they escaped, or were rescued, or died.

3.12. **Write the `main()` function**

Your `main()` function must declare an `Escape` object and call its `runEscape()` function. The entire program control flow must be implemented in the `Escape` object, and `main()` must do nothing else.

3.13. **Packaging and documentation**

3.13.1. Your code must be correctly separated into header and source files, as we saw in class.

3.13.2. You must provide a `Makefile` that separately compiles each source file into a corresponding object file, then links all the object files to produce the program executable. Your `Makefile` must also include the `clean` target that removes all the object files and the program executable. **Do not** use an auto-generated Makefile; it must be specific to this program.

3.13.3. You must provide a plain text `README` file that includes:

(a) a preamble (program author, purpose, list of source, header, and data files, if applicable)

(b) compilation and launching instructions, including any command line arguments

3.13.4. Use either the `tar` or the `zip` utility in Linux to package into one `.tar` or `.zip` file **all the files** required to build and execute your program.

3.13.5. Do not include any additional files or directories in your submission.

3.13.6. All class definitions must be documented, as described in the course material, section 1.3. Please **DO NOT** place inline comments in your function implementations.

3.14. **Integration with Assignment #4 code**

To earn the full 5 bonus marks for integration, your program must fully meet all of the criteria found in all the instructions for both assignments, and it must execute and print all data perfectly.

This means that all features from Assignment #4 must be integrated into the Assignment #5 program. Where there are conflicting instructions between the two assignments, the more advanced option must be implemented.

4. Constraints

Your design and implementation must comply with all the rules of correct software development and programming conventions that we have learned during the course lectures, including but not restricted to:

4.1. The code must be written using the C++11 standard that is supported by the course VM, and it must compile and execute in that environment. It must not require the installation of libraries or packages or any software not already provided in the course VM.

4.2. Your program must comply with the principle of least privilege, and it must follow correct encapsulation principles, including the separation of control, view, entity, and collection object functionality.

4.3. Your program must not use any library classes or programming techniques that are not used in the in-class coding examples. Do not use any classes, containers, or algorithms from the C++ standard template library (STL), unless explicitly permitted in the instructions.

4.4. If base code is provided, do not make any unauthorized changes to it.

4.5. Your program must follow basic OO programming conventions, including the following:

4.5.1. Do not use any global variables or any global functions other than `main()`.

4.5.2. Do not use `struct`s. You must use classes instead.

4.5.3. Objects must always be passed by reference, never by value.

4.5.4. Functions must return data using parameters, not using return values, except for getter functions and where otherwise permitted in the instructions.

4.5.5. Existing functions and predefined constants must be reused everywhere possible.

4.5.6. All basic error checking must be performed.

4.5.7. All dynamically allocated memory must be explicitly deallocated.

4.6. You may implement helper classes and helper member functions, but only if the design is consistent with the provided instructions. Design and implementation choices that undermine the learning outcomes will not earn any marks.

5. Submission Requirements

5.1. You will submit in *Brightspace*, before the due date and time, one `tar` or `zip` file that includes all your program files, as described in the **Packaging and documentation** instructions.

5.2. Late submissions will be subject to the late penalty described in the course outline. No exceptions will be made, including for technical and/or connectivity issues. Do not wait until the last day to submit.

5.3. Only files uploaded into *Brightspace* will be graded. Submissions that contain incorrect, corrupt, or missing files will be graded as is. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading Criteria

- 45 marks: code quality and design
- 30 marks: coding approach and implementation
- 20 marks: program execution
- 5 marks: program packaging
- *5 marks*: *bonus marks for full integration with Assignment #4 code*