COMP 2404 B/C – Assignment #4

Due: Thursday, March 21 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). Timmy and Harold were once again attempting to rescue a baby Dragon from the evil wizard. Our heroes were caught before they could complete their mission, and they were taken prisoner and thrown into the Snorc Pit. Now, Timmy and Harold are attempting to escape by climbing out of the Pit.

Your program will simulate the attempted escape by our heroes out of the Snorc Pit, as the snorcs try to bite and stop them. It will use polymorphism to model the randomized behaviours of different participants (both heroes and snorcs). Because the movements of each participant are randomly determined, every execution of the simulation will have a different outcome. Your code must display the simulation as it progresses, including the changing positions of each participant, in the terminal window. It must print out the outcome of the simulation at the end, specifically whether the heroes escaped the Snorc Pit or died in the attempt.

2. Learning Outcomes

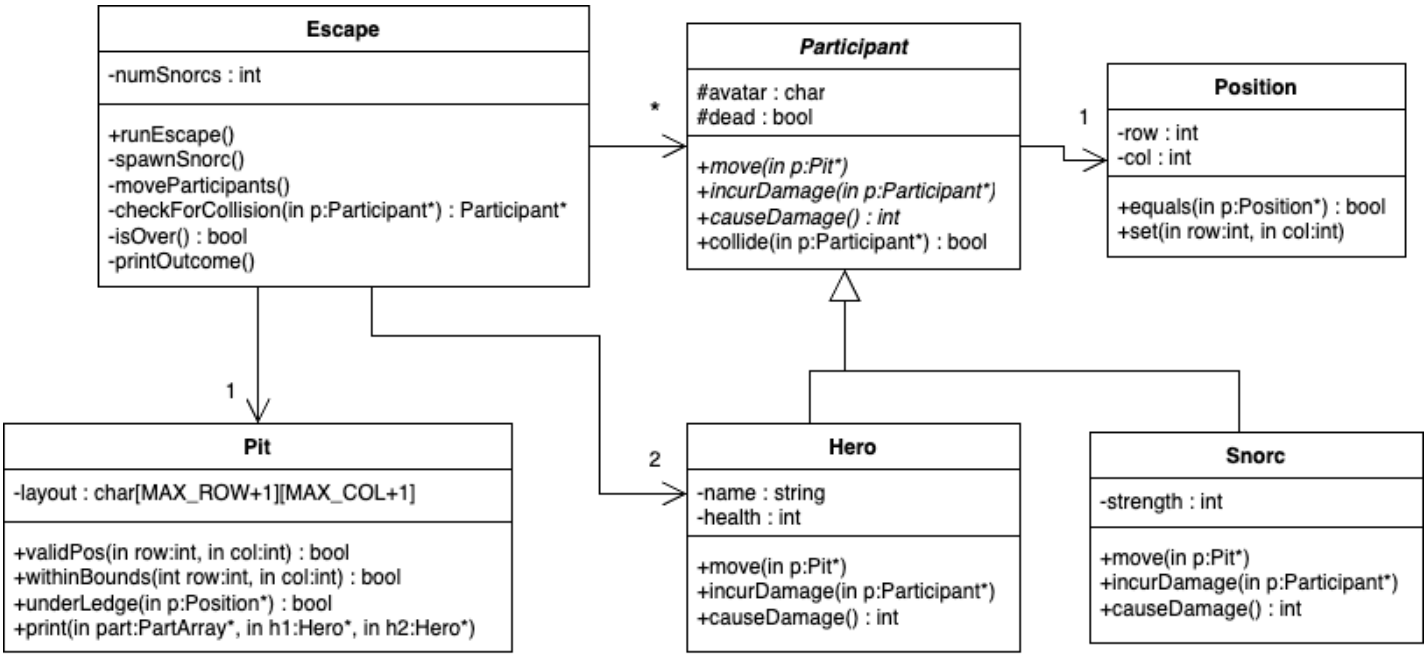
With this assignment, you will:

- practice implementing a design that is given as a UML class diagram
- apply the OO concepts of inheritance and runtime polymorphism through dynamic binding
- work with virtual and pure virtual functions in C++

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 `a4-posted.tar` file contains the required constant definitions and a skeleton `Escape` class with the exact Snorc Pit layout that your program must use, *without modification*.

A global `random()` function has also been provided for you, 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.

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. The layout of this grid is provided in the base code, and you must use it **exactly**. In addition to showing the walls at the sides of the Pit, the layout also contains rocky ledges sticking out of the walls. Heroes *cannot* climb over the ledges, so they must go around them instead. Snorcs *can* climb directly over ledges, but they cannot climb higher than the bottom seven rows of the PIt.

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

3.3.5. 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.

3.3.6. All participants in the escape, both heroes and 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.

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

3.3.8. The simulation begins with no snorcs in the Pit. At every iteration of the game 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.

3.3.9. At every iteration of the game loop, for every participant (hero and spawned 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) *all position rows and columns must be valid within the Pit*; if a new row or column is computed below zero, then it is repositioned at zero; if a new row or column is computed beyond the maximum, then it is reset to that maximum

(c) 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)

(d) if a collision has occurred, it must be handled **polymorphically**; if two heroes or two snorcs collide with each other, nothing happens; if a hero and a snorc collide, the snorc bites the hero, and the hero's health must be decreased by the amount of strength possessed by that snorc

3.3.10. At the end of every iteration of the game loop, the Pit must be printed out, as well as both heroes' names and health indicators next to the bottom two rows of the Pit, as seen in the workshop video.

3.3.11. Once the game loop has concluded, the Snorc Pit and both hero names and health indicators must be printed to the screen one final time, and the outcome of the escape must be printed out. The program must state either that both heroes have escaped, or that both heroes are dead, or it must indicate which hero escaped the Pit and which one died.

3.4. **Understand the participant movements**

3.4.1. Each move by a hero is computed as follows:

(a) if the hero is dead or has already escaped, they do not move

(b) if the hero is currently located underneath a ledge:

(i) they stay in the same row

(ii) they moves from their current column by a randomly determined one column to the left, or one column to the right

(c) if the hero is currently not located underneath a ledge:

(i) they move from their current row in accordance with Table 1

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

Table 1: Hero vertical movements

Type of move	Probability	What happens
Slow climb	50%	move 1 row up
Fast climb	10%	move 2 rows up
Slide	40%	move 1 row down

3.4.2. Each move by a snorc is computed as follows:

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

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

(c) 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

3.5. **Implement the `Position` class**

The `Position` class represents the position that a participant is occupying in the Snorc Pit. It contains the data members and member functions indicated in the UML class diagram. In addition:

3.5.1. The default constructor must take two parameters and initialize the corresponding data members.

3.5.2. The `set()` member function must set the row and column to the given parameters.

3.5.3. Any function that sets the position's row and column must perform validation. If the parameter row or column is outside the boundaries of the Snorc Pit, it must be reset to the inside edge of the Pit.

3.5.4. The `equals()` member function must compare both rows and columns.

3.5.5. The program requires getter member functions for both row and column.

3.6. **Implement the `Pit` class**

The `Pit` class represents the Snorc Pit in the simulation. It contains the data members and member functions indicated in the UML class diagram. In addition:

3.6.1. The data member represents the *empty* Snorc Pit layout, as provided in the skeleton `Escape` class in the base code. It **cannot** be used to store participant avatars, at any time during the simulation.

3.6.2. The constructor takes a given layout as parameter, and it must initialize the data member accordingly. Remember, the assignment operator cannot be used to copy entire arrays.

3.6.3. The `withinBounds()` member function determines whether or not the given row and column represent a valid position, possibly a ledge, within the boundaries of the Pit.

3.6.4. The `validPos()` member function determines whether or not the given row and column represent a valid position, but *not* a ledge, within the boundaries of the Pit.

3.6.5. The `underLedge()` member function determines whether or not a given position is located directly underneath a ledge in the layout.

3.6.6. The `print()` member function must do the following:

(a) declare a temporary 2D grid of characters to contain the Pit layout and the participant avatars

(b) initialize the temporary grid with the contents of the empty layout data member

(c) loop through the given participants collection, 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 and health indicators of each hero must be printed out

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

3.7. 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 two derived classes, `Hero` and `Snorc`, must be *concrete*.

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

3.7.1. The default constructor must take three parameters: a participant's avatar, and its initial row and column. All data members must be initialized, and a new `Position` must be dynamically allocated and initialized for the participant.

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

3.7.3. The `move()` member function must be **pure virtual**. It will be used in the simulation to compute successive new positions for a participant.

3.7.4. 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.

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

3.7.6. The `collide()` member function must determine whether or not the participant is currently located at the same position as the parameter participant.

3.7.7. You will need getter member functions for the avatar, as well as the position row and column. You will implement a `bool isDead()` member function that determines if the participant is dead or not. An additional `bool isSafe()` member function must be implemented to check whether or not the participant has climbed out of the Pit.

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

3.8. Implement the `PartArray` class

You will copy the `Array` class that we implemented in the coding example of section 2.2, program #1, into a new collection class called `PartArray`. Then, you must modify the class so that it stores `Participant` object pointers as data, and you must remove the `print()` member function.

You will implement a `Participant* get(int)` member function that returns the element at the given index, and you will provide a getter member function for the current number of array elements.

3.9. 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 the data members and member functions indicated in the UML class diagram. In addition:

3.9.1. The default constructor must take four parameters: the hero's avatar, initial row and column, and name. It must initialize all data members using these parameters, including calling the base class constructor using *base class initializer syntax*.

3.9.2. The virtual `move()` member function must compute a new position for the hero, in accordance with the escape rules described in instruction 3.4.1. If the computed position is valid (within bounds and not a ledge position), the hero's current position is updated accordingly. If the computed position is invalid, the hero does not move.

3.9.3. 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) the hero's health must be reduced by the amount of damage caused by the parameter participant

(c) 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

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

3.9.5. You will need getter member functions for the hero's name and health indicator.

3.10. Implement the `Snorc` class

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

3.10.1. 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.

3.10.2. The virtual `move()` member function must compute a new position for the snorc, in accordance with the escape rules described in instruction 3.4.2. If the computed position is within bounds, the snorc's current position is updated accordingly. If the computed position is invalid, the snorc does not move.

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

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

3.11. 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 `PartArray` object. All participants in the escape **must** be stored in this collection, otherwise polymorphism will not work.

3.11.2. The pit must be stored as a pointer to a `Pit` object.

3.11.3. 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.

3.11.4. 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

(c) dynamically allocate a new pit with the given layout as parameter

3.11.5. The destructor must deallocate the required memory.

3.11.6. The `runEscape()` member function implements the game loop, exactly as described in the escape rules in instruction 3.3, and it prints the result of the simulation.

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

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

3.11.9. The `checkForCollision()` member function loops over the participants collection 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 loops over the participants collection, and for each participant, it does the following:

(a) use dynamic binding to *polymorphically* move the participant to a new position in the Pit

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

(c) if there is a collision, the two participants **both** *polymorphically* incur damage from each other

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

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.

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

- 40 marks: code quality and design
- 35 marks: coding approach and implementation
- 20 marks: program execution
- 5 marks: program packaging