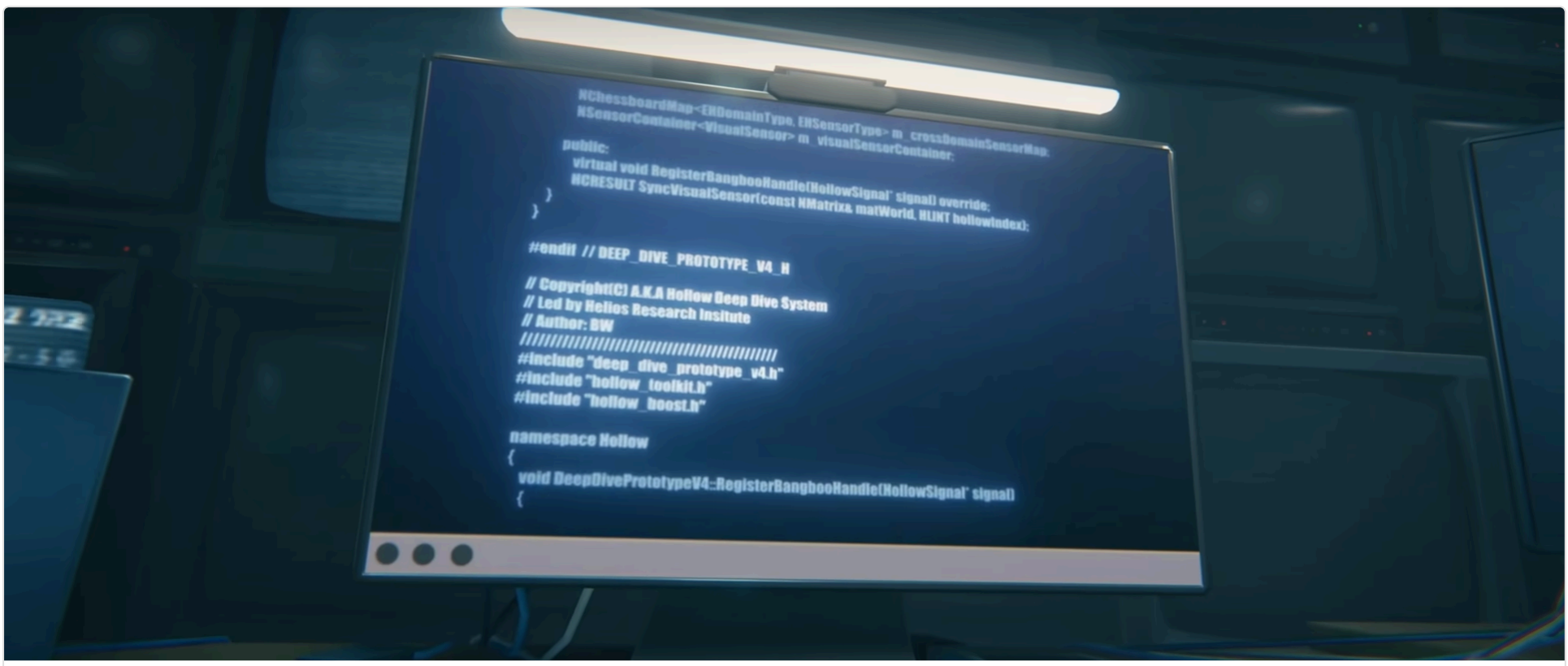
COMP 2012 Object-Oriented Programming and Data Structures

Assignment 2 Zenless Zone Zeta



Source: https://www.bilibili.com/video/BV1QE421A7Y9

Introduction

The author of this assignment has been obsessed with a game called Zenless Zone Zeta recently. One day, he watched a video and realised that C++ actually appeared in the game!... As a lover of the game and C++, he decided to recreate part of the game, using C++. Could you help him fulfill his wish?

Settings

In the game, the city is facing a threat named *Hollow*, which is a space that will randomly appear out of nowhere in the real world. Since the space inside Hollow is extremely dangerous, the main character (you, as the player) has invented a device - HDD. Using the HDD, you will be able to possess the *BangBoo* (i.e. a robot) and control it to explore the Hollow instead. The goal of this assignment is to complete coding the HDD.

TLDR: The assignment is about coding a maze (Hollow), allowing the player to control the robot (BangBoo) to the goal.

We value academic integrity very highly. Please read the Honor Code section on our course webpage to make sure you understand what is considered as plagiarism and what the penalties are. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there will be additional disciplinary actions from the department and university, upto and including expulsion.

End of Introduction

Description

Please read the FAQ and Change Log sections regularly, and do check it one day before the deadline to make sure you don't miss any clarification, even if you have already submitted your work by that time. You can also raise any questions on Piazza and remember to use the "pa2" tag.

Code structure

The skeleton code structure is as follows:

```
PA2
├── hollow.h
├── hollow.cpp
├── entity.h
├── blockEntity.h
├── blockEntity.cpp
├── movableEntity.h
├── movableEntity.cpp
├── fairy.h
├── utils.h
├── main.cpp
├── main_test.cpp
├── Makefile
└── input.txt
```

The `hollow.h` contains the `Hollow` class declarations and implementations for some given functions. More details about the `Hollow` class will be explained below.

The `hollow.cpp` should be submitted by implementing the TODO functions marked in `hollow.h`.

The `entity.h` contains the `Entity` class declarations, where the `Entity` class serves as an abstract base class for classes `BlockEntity` and `MovableEntity`.

The `blockEntity.h` contains the `BlockEntity` class declarations and implementations for some given functions. It also contains definitions for the subclasses of `BlockEntity`. More details about the `BlockEntity` class will be explained below.

The `blockEntity.cpp` should be submitted by implementing the TODO functions marked in `blockEntity.h`.

The `movableEntity.h` contains the `MovableEntity` class declarations and implementations for some given functions. It also contains definitions for the subclasses of `MovableEntity`. More details about the `MovableEntity` class will be explained below.

The `movableEntity.cpp` should be submitted by implementing the TODO functions marked in `movableEntity.h`.

The `fairy.h` contains the `Fairy` class declarations and definitions. In the game, *Fairy* is the name of a manager AI helping the main character. In the code, it serves the purpose of managing memory allocation of the `Hollow` object, and player interaction.

The `utils.h` contains some utility class & enum.

The `main.cpp` contains the `main` function.

The `main_test.cpp` contains the `main` function for running test cases. Please refer to ZINC or test case results for the expected output and perform necessary comparisons. You can find the test case ID either on ZINC or in the filenames of the test case results.

The `Makefile` and `input.txt` are for you to compile your program and set up the maze input file. More details will be given at the Resource & Sample Program section.

*** It is strongly recommended to read ALL the `.cpp/.h` files after reading this site to better understand the program logistics.**

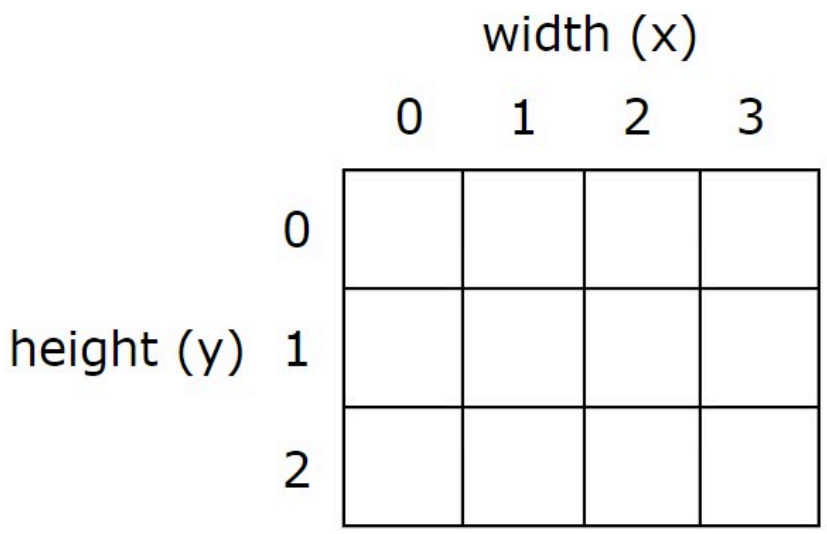
Hollow

The following is a part of the `Hollow` class definition in this assignment, which is defined in `Hollow.h`.

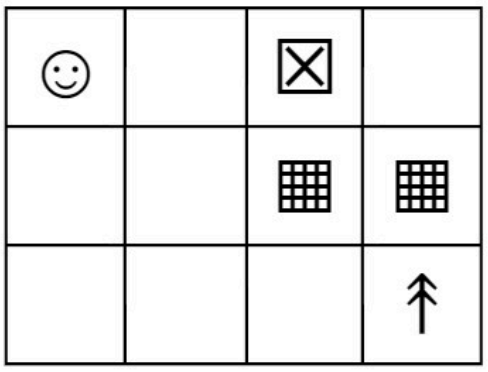
```
class Hollow {
    private:
        // Properties of the "grid"
        int levels, height, width;
        // Dynamic 3D array of BlockEntity*
        BlockEntity**** grid;
        // Storing the pointer of bangboo
        BangBoo* bangboo = nullptr;
        // Whether the game is ended (won)
        bool gameWon = false;
};
```

Simply speaking, the `Hollow` class stores a 3D maze where the player will control the BangBoo to explore, where the maze data is stored in the `grid` which is a dynamic 3D array.

The `grid` may have many `levels`, with each level being a 2D maze that has the same `height` and `width`. The following diagram shows an example of the 2D maze (height 3, width 4) at a level.



The coordinate system of the 2D maze at a level



An example of a 2D maze, the meaning of the icons will be explained later

BlockEntity

From above, it can be inferred that the entries of the 3D maze are `BlockEntity*`, which are pointers to `BlockEntity` objects. The `BlockEntity` class is an abstract base class that is inherited by block (i.e., immovable) entities. That means, when these entities are created and registered with the `Hollow`, the location of these entities will never be changed. The subclasses of these entities include:

- `Empty`
  - The default `BlockEntity` object when the `Hollow` is being initialized.
- `Wall` (Icon: ▦)
  - A block entity that stops the bangboo from moving.
- `Goal` (Icon: G)
  - The goal of the Hollow, the player wins when the bangboo steps on it.
- `Portal` (Icon: ↟ for ascend; ↡ for descend)
  - The bangboo will ascend to the next level (e.g. from 0 to 1) if stepped on ↟, while keeping the same xy-coordinate. Same for ↡ but the bangboo will descend to the previous level (e.g. from 1 to 0).
- `Door` (Icon: ⍈ for locked; ⬚ for unlocked)
  - When a door is locked, it is the same as a `Wall`. Each door is associated with some `Switch` (1 at least; 5 at most), and the door will be unlocked when all switches are in the "on" state.
- `Lever` & `Button` (Icon: ◈ for on; ◇ for off)
  - `Lever` and `Button` are both subclasses of `Switch`, since `Switch` is an abstract base class. They are at the "off" state by default and they will also be associated with some `Door` (0 at least; 5 at most). They will be turned into "on" state when a `MovableEntity` steps on it.
    The difference between `Lever` and `Button` is: `Button` will turn back to the "off" state when there is no `MovableEntity` stepping on it but the `Lever` will never be off once turned on.

Also, there is a `MovableEntity*` attribute in the class `BlockEntity`. This serves the purpose that `MovableEntity` can step on `BlockEntity`, and also implies that each `BlockEntity` can only have at most one `MovableEntity` being stepped on it. If there is no `MovableEntity` stepping on it, it would be `nullptr`.

MovableEntity

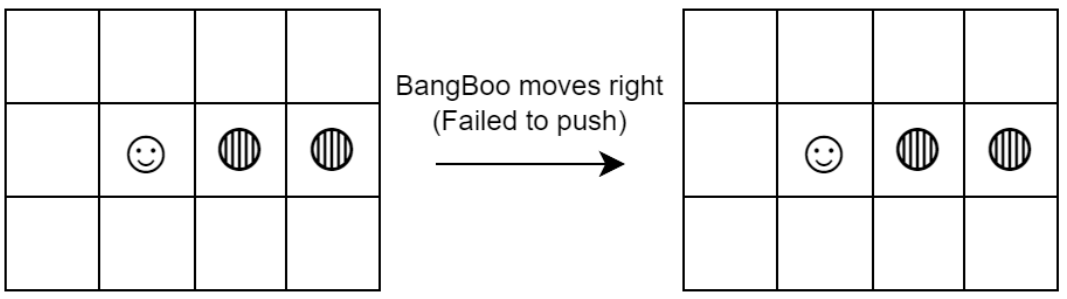
The `MovableEntity` class is an abstract base class that is inherited by movable entities. That means the location of these entities can be changed during the game. The subclasses of these entities include:

- `BangBoo` (Icon: ☺)
  - The robot that the player will control during the game. Each `Hollow` will only be associated with one `BangBoo`.
- `Stone` (Icon: ◍)
  - A `Stone` can be pushed by the `BangBoo`, or chain pushed by another `Stone`. Note that it can't be pushed once it reaches a `Wall` or the boundary. The following is an example of some cases:

Case 1:

Case 2:

Case 3:



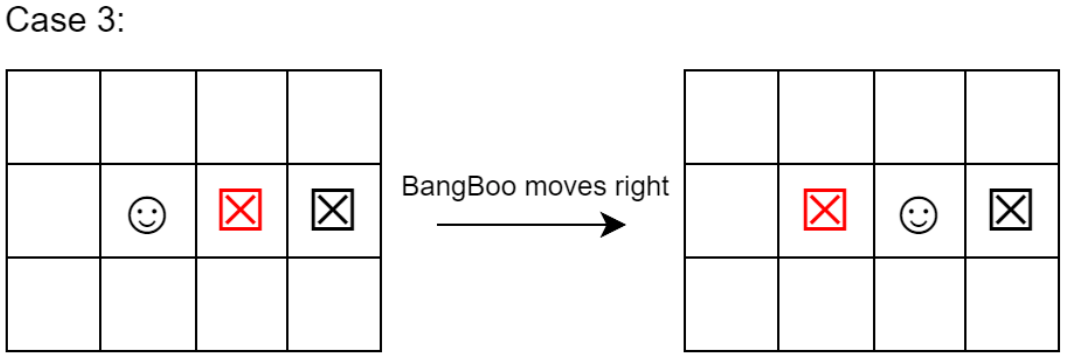
- `Box` (Icon: ⊠)
  - A `Box` is similar to a `Stone`, with the exception that a `Box` can swap with the `BangBoo`: if the `BangBoo` is DIRECTLY pushing it, and the `Box` cannot be pushed forward. Here are some examples (The `Box` denoted in red is being

directly pushed by `BangBoo`, so it swaps with the `BangBoo`):

Case 1:

Case 2:

Case 3:



Additional Remarks

- You are required to implement the corresponding function in the correct `.cpp` file. The required header files are already included at the start of the three `.cpp` files which you should NOT modify, **and submit only these three files to ZINC**.
- You are NOT allowed to include any additional libraries. All required libraries are already included in the skeleton code.
- You are NOT allowed to use the `auto` keywords, **`static_cast`**, **`dynamic_cast`**, **or `typeid`**.
- You are NOT allowed to define any static or global variables, or any additional classes or structs. There is no need for it in this assignment.

End of Description

Tasks

This section describes the functions that you will need to implement. It is suggested that you complete the parts in order.

Part I: Hollow

Implement the following functions of `Hollow` class in the file `Hollow.cpp`.

```
Hollow(int levels, int height, int width)
```

Description - The other constructor of `Hollow`. This will be used to create the base maze.

- Initialize the corresponding member variables.
- Be careful with memory allocation when initializing the `grid`, you should initialize the `BlockEntity` to `Empty`.
- Please refer to the `getBlockEntityFromLoc` function, and the 2D maze coordinate system mentioned above for indexing the `grid`.

Parameters

- `levels` - The number of levels in the `Hollow`, must be at least 1.

- `height` - The height of a level in the `Hollow`, must be at least 1.
- `width` - The width of a level in the `Hollow`, must be at least 1.

```
~Hollow()
```

Description - The destructor of `Hollow`. Remember to deallocate any memory that was allocated.

```
BlockEntity* registerBlockEntity(const BlockEntity& bentity)
```

Description - Used to put the corresponding `BlockEntity` onto the `grid`.

- The function will only be called before the start of the game, you may refer to `Fairy::readHollowFromFile` for its usage.
- You may want to refer to `blockEntity.h` for assistance with this task.
- You may assume that every registration is valid - the same location cannot be registered twice for any `Entity`.

Parameters

- `bentity` - the block entity to be registered to the `grid`.

Return value - The pointer to the `BlockEntity` assigned to the `grid`.

```
MovableEntity* registerMovableEntity(const MovableEntity& mentity)
```

Description - Used to place the corresponding `MovableEntity` (except `BangBoo`) onto the `grid`.

- The function will only be called before the start of the game, you may refer to `Fairy::readHollowFromFile` for its usage.
- You may want to refer to `movableEntity.h` for assistance with this task.
- You may assume that every registration is valid - the same location cannot be registered twice for any `Entity`.

Parameters

- `mentity` - the movable entity to be registered to the `grid`.

Return value - The pointer to the `MovableEntity` assigned to the `grid`.

```
BangBoo* registerBangBoo(const BangBoo& bangboo)
```

Description - Used to place the corresponding `BangBoo` onto the `grid`.

- The function will be called at least once before the start of the game, you may refer to `Fairy::readHollowFromFile` for its usage.
- You may want to refer to `movableEntity.h` for assistance with this task.
- You may assume that every registration is valid - the same location cannot be registered twice for any `Entity`.
- Remember to also initialize the member variable.
- (Added on 4th Nov) This function does nothing for any subsequent calls after the first call of this function, and it should return `nullptr` in that case.

Parameters

- `bangboo` - the bangboo to be registered to the `grid`.

Return value - The pointer to the `BangBoo` assigned to the `grid`.

```
bool locInHollow(const Location& loc) const
```

Description - Determines whether the given `Location` is valid (i.e. not out of bounds) in the `Hollow`.

Parameters

- `loc` - the `Location` to be checked.

Return value - Returns true if the given `Location` is in the `Hollow`.

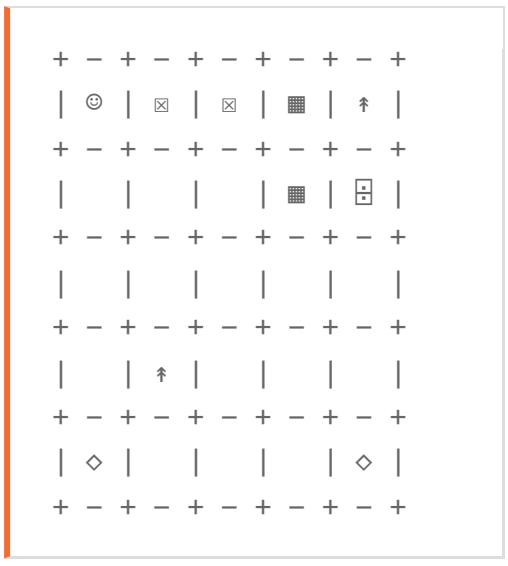
```
void print() const
```

Description - The print function of the `Hollow` class.

- This function prints the 2D maze of the level where the `bangboo` is currently at.
- It prints the grid using the following:
  - "`+`" string for the corners of each cell
  - "` - `" string for the horizontal edges of each cell
  - "`|`" string for the vertical edges of each cell
  - "` X `" string for the content in the cell, where `X` is the icon representation of the `BlockEntity`, or `MovableEntity`. The function for icon representation has already been implemented for you, just call the `BlockEntity::toString()` function, it will print the `MovableEntity` over `BlockEntity`.

Notes

- The following is an example output of the `print` function, you can try the sample program yourself to test if you are unsure.
  (Ideally, the output grid should be perfectly if you use the VSCode Terminal or a Unix Terminal.)



Part II: BlockEntity's onStep & onLeave

Header of `blockEntity.h`

You will notice that there are two functions, namely `onStep(MovableEntity* mentity)` and `onLeave(MovableEntity* mentity)`. They are intended to be overridden by subclasses of `BlockEntity` to perform block-specific updates, which will be called when it is stepped on or left by the `mentity`.

You should complete the following functions and implement them in `blockEntity.cpp`.

```
void Switch::onStep(MovableEntity* mentity)
void Button::onLeave(MovableEntity* mentity)
```

Description - Change the status and update the associated doors when the `Switch` or `Button` is being stepped on or left.

- Any `MovableEntity` should be able to activate the switch.

Parameters

- `mentity` - the movable entity stepped/left.

```
void Portal::onStep(MovableEntity* mentity)
```

Description - Teleport the movable entity to ascend or descend one level according to the portal type, if the movable entity is a `BangBoo`.

- The BangBoo should be teleported to the same xy-coordinate.
- If the teleport destination already contains a movable entity, destroy that movable entity.

Parameters

- `mentity` - the movable entity stepped/left.

Notes

- You may assume that all teleportation is valid (e.g. there are enough levels for the bangboo to ascend/descend, the BangBoo will not be teleported to a `Wall`, etc.)
- You don't need to call the `onStep` function at the destination location after teleportation.
- It is suggested that you use the given utility function `Hollow::moveMentityFromTo` directly.

```
void Goal::onStep(MovableEntity* mentity)
```

Description - Set the `Hollow` to the win state if the movable entity that steps on is the bangboo.

Parameters

- `mentity` - the movable entity stepped.

Part III: BlockEntity & MovableEntity's update

Game Update Schematics

It can be seen that when there is a movement input for the `BangBoo`, the game starts to update from the code:

```
class Hollow {
    public:
        // ...
        void moveUpdate(Direction toDir) {
            getBlockEntityFromLoc(this->bangboo->loc)->update(nullptr, toDir);
        }
};
```

For every `Entity` object, its `bool update(MovableEntity* fromEntity, Direction toDir)` function is being defined for the push update. In the function, the `fromEntity` represents the movable entity that wants to step into the current location (`nullptr` if it's from the player's input), and `toDir` indicates the direction to which the current entity is being pushed.

If the current location is already steppable, the function will return `true` immediately without performing any actions. However, if the current location is not steppable because a movable entity blocks it, the function should attempt to make the current location steppable by calling the `update` function of the other `Entity`. If the location then becomes steppable, the function will return `true`; otherwise, it will return `false`.

You should complete the following functions and implement them in `blockEntity.cpp` and `movableEntity.cpp`.

Notes

- You should move the entities and call the `onStep` and `onLeave` functions in `movableEntity.cpp` only, but **NOT** in `blockEntity.cpp`. Since, according to the principle of object-oriented programming, how a movable entity moves should be defined by the movable entity itself.
- Again, it's suggested that you use the given utility function `Hollow::moveMentityFromTo` when moving the movable entities.

```
bool BlockEntity::update(MovableEntity* fromEntity, Direction toDir)
```

Description - The update function for generic block entities.

Parameters

- `fromEntity` - the movable entity that is trying to step into the current location.
- `toDir` - the direction to which the current entity is being pushed.

Return value - If the location of this `BlockEntity` is steppable after the update.

```
bool Wall::update(MovableEntity* fromEntity, Direction toDir)
bool Door::update(MovableEntity* fromEntity, Direction toDir)
```

Description - The update function for specific block entities `Wall` and `Door`.

- Refer to the description section for the expected update behavior of these block entities.

Parameters

- `fromEntity` - the movable entity that is trying to step into the current location.
- `toDir` - the direction to which the current entity is being pushed.

Return value - If the location of this `BlockEntity` is steppable after the update.

```
bool MovableEntity::update(MovableEntity* fromEntity, Direction toDir)
```

Description - The update function for generic movable entities.

- Think of how you should update the movable entity.
- You should also call the `onStep` and `onLeave` functions accordingly.

Parameters

- `fromEntity` - the movable entity that is trying to step into the current location.
- `toDir` - the direction to which the current entity is being pushed.

Return value - If the initial location of this `MovableEntity` is steppable after the update.

```
bool Box::update(MovableEntity* fromEntity, Direction toDir)
```

Description - The update function for the specific movable entity `Box`.

- Refer to the description section for the expected update behavior of the `Box`.
- You should perform the swapping between the `Box` and `BangBoo` in this function if applicable. The function should return `false` after swapping, as the original location is still not steppable after the update.
- You should also call the `onStep` and `onLeave` functions accordingly.

Parameters

- `fromEntity` - the movable entity that is trying to step into the current location.
- `toDir` - the direction to which the current entity is being pushed.

Return value - If the initial location of this `MovableEntity` is steppable after the update.

End of Tasks

Resources & Sample program

- Skeleton code: pa2-skeleton.zip
- Demo program:
  - Linux: pa2

Compiling the Program

You can compile the program by running the `make pa2` command directly with the given Makefile, it should work on all operating system. You can also compile the test program by running the `make pa2_test` command.

Custom Maze Design

The program will output error if there doesn't exist a valid maze file `input.txt` in the same directory of the executable. A default `input.txt` has already been given to you. If you would like to change the maze setup to test out the expected behavior, feel free to do so. Here is the file format of the maze `input.txt`:

- 1st line of the file: `l w h dsconn`, where `l`, `w`, `h` are the levels, width, and height of the Hollow respectively. The `dsconn` integer represents the number of PAIRS of connection between `Door` and `Switch`.
- The next `h` lines: Each line contains `w` characters, which denotes the `Entity` at that location. Read the code section `fairy.h (Line 40-70)` for details on the character-`Entity` mapping.
- The above input will repeat for `l` times, until the 2D maze of all levels has been read.
- The next `dsconn` lines: Each line will be in format of `dl dx dy sl sx sy`, where `(dl, dx, dy)` is the `Location` of the `Door`, and `(sl, sx, sy)` is the `Location` of the `Switch` to be connected to the `Door`.

Test cases

There are 17 given test cases available on ZINC, and can be also found in the `main_test.cpp` file. Initially, these 16 test cases are executed without any memory leak checking. Then, the same 16 test cases are run again with their original IDs incremented by 17. For example, test case #13 is repeated as test case #29.

You can find the inputs and expected outputs of the 16 given test cases in the `public_cases` folder in above skeleton code zip. Your implementation should aim to produce the same output as the provided sample output for all given test cases above.

It's important to note that the sample output cannot cover all possible cases, so it is part of the assessment for you to design your own test cases to thoroughly test your program's functionality.

In addition to the give test cases, there will be additional test cases which won't be revealed to you before the deadline. They are referred as "hidden test cases" below.

End of Resources & Sample I/O

Submission & Grading

Deadline: 9 November 2024 Saturday HKT 23:59.
Submit a zip containing three files: `hollow.cpp`, `blockEntity.cpp` and `movableEntity.cpp` to ZINC.

Notes:

- You may submit your files multiple times, but only the latest version will be graded.
- Submit early to avoid any last-minute problem. Only ZINC submissions will be accepted.
- The ZINC server will be very busy in the last day especially in the last few hours, so you should expect you would get the grading result report not-very-quickly. However, as long as your submission is successful, we would grade your latest submission with all test cases after the deadline.

Compilation Requirements

It is **required** that your submissions can be compiled and run successfully in our online autograder ZINC. If we cannot even compile your work, it won't be graded. Therefore, for parts that you cannot finish, just put in dummy implementation so that your whole program can be compiled for ZINC to grade the other parts that you have done. Empty implementations can be like:

```
int SomeClass::SomeFunctionICannotFinishRightNow()
{
    return 0;
}

void SomeClass::SomeFunctionICannotFinishRightNowButIWantOtherPartsGraded()
{
}
```

Please make sure your code submitted to ZINC can be compiled successfully in the score report.

Grading Scheme

The grading will be separated into test cases, which may test individual functions or the whole program flow. Therefore, try to finish as many tasks/functions as you can.

On ZINC, test case **#1XY** is the `-fsanitize` version of `#XY`. If your program has correct output but also has runtime errors including memory leak, you will lose points for the `-fsanitize` test case versions.

About Memory Leak and Other Potential Errors

Memory leak checking is done via the `-fsanitize=address,leak,undefined` option (related documentation here) of a recent g++ compiler on Linux (it won't work on Windows for the versions we have tested). Check the "Errors" tab (next to the "Your Output" tab in the test case details popup) for errors such as memory leaks. Other errors/bugs such as out-of-bounds, use-after-free bugs, and some undefined-behavior-related bugs may also be detected. You will get a 0 mark for the test case if there is any error. Note that if your program has no errors detected by the sanitizers, then the "Errors" tab may not appear. If you wish to check for memory leaks yourself using the same options, you may follow the Checking for memory leak yourself guide.

Late Submission Policy

There will be a penalty of -1 point (out of a maximum of 100 points) for every minute you are late. For instance, since the deadline for assignment 1 is 23:59:00 on **Oct 12**, if you submit your solution at 1:00:00 on **Oct 13**, there will be a penalty of -61 points for your assignment. However, the lowest grade you may get from an assignment is zero: any negative score after the deduction due to a late penalty (and any other penalties) will be reset to zero.

End of Submission & Grading

Frequently Asked Questions

Q: My code doesn't work, here it is, can you help me fix it?
A: As the assignment is a major course assessment, to be fair, you are supposed to work on it by yourself and we should never finish the tasks for you. We are happy to help with explanations and advice, but we are **not allowed** to directly debug for you.

Q: Am I allowed to create helper functions?
A: Yes.

End of FAQ

Change Log

- **4th November 02:10**
  1. Added description for `Hollow::registerBangBoo`:
     This function does nothing for any subsequent calls after the first call of this function, and it should return `nullptr` in that case.

End of Change Log