Project 3

Lemmings

For questions about this project, first consult your TA.
If your TA can't help, ask Professor Nachenberg.

Part 1 due: Monday, February 23
Part 2 due: ~~Monday, March 2~~ Tuesday, March 3

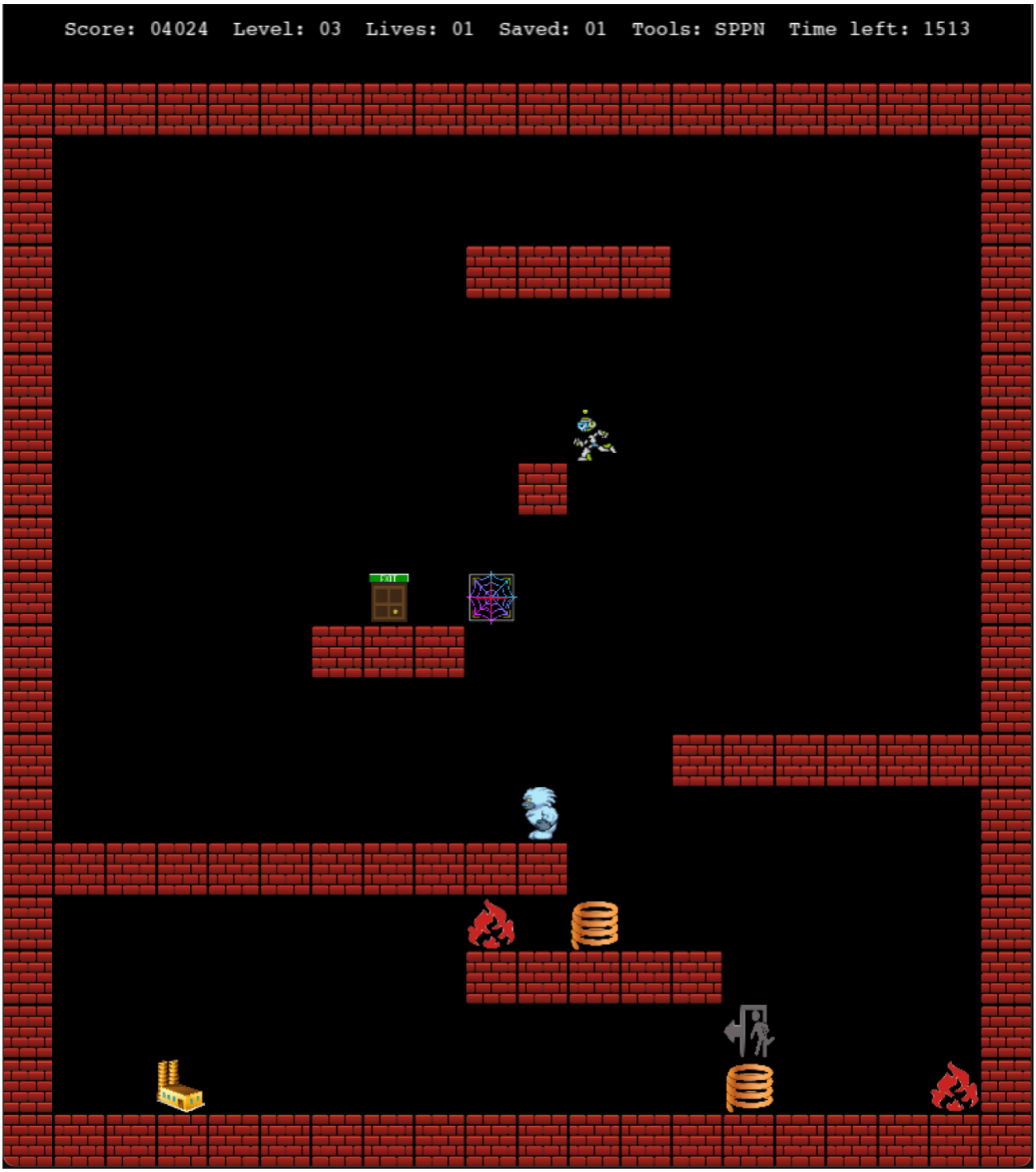
WHEN IN DOUBT ABOUT A REQUIREMENT, YOU WILL NEVER LOSE CREDIT IF YOUR SOLUTION WORKS THE SAME AS OUR POSTED SOLUTION.
SO PLEASE DO NOT ASK ABOUT ITEMS WHERE YOU CAN DETERMINE THE PROPER BEHAVIOR ON YOUR OWN FROM OUR SOLUTION!

PLEASE THROTTLE THE RATE YOU ASK QUESTIONS
TO 1 EMAIL PER DAY! IF YOU'RE SOMEONE WITH
LOTS OF QUESTIONS, SAVE THEM UP AND ASK ONCE.

Table of Contents

Introduction

In the game of Lemmings, your job is to get the lemmings from their starting point safely to their exit.



Each level is a small 20×20 grid “world” that contains terrain and hazards (mostly floor bricks, plus roaming ice monsters and bonfires), along with a lemming factory that steadily produces

lemmings. Your lemmings are not directly controlled. Instead, you control a builder cursor that can move freely around the grid and place a limited number of tools onto empty squares in order to influence how lemmings travel.

The big idea is that this is a simulation puzzle. Lemmings will keep walking, falling, bouncing, and climbing based on a handful of simple rules. Most levels are not solvable “as is,” because lemmings will fall too far, get burned, or otherwise fail to reach the exit often enough. Your goal is to intervene by placing the right tools in the right places, while working within a limited tool inventory and a fixed time limit.

A level attempt ends in one of two ways: you either complete the level by saving enough lemmings, or you lose a life because you ran out of time, gave up, or lost too many lemmings (by certain kinds of deaths). If you still have lives left, you can replay the level from scratch; if not, the game ends. If you finish a level, the game advances to the next level file.

In essence, each level is a race against both the environment and the clock. Lemmings are spawned over time by the lemming factory, and you must guide them to the exit. You begin the game with three lives. You lose one life immediately if you give up on the level attempt, if the timer reaches zero before you have saved enough lemmings, or if too many lemmings die in ways that count as “deaths” for the level’s failure condition.

The lemming population for every level is fixed. Every level spawns exactly 10 lemmings. You must save at least half of them to complete the level, which means you must save 5 or more lemmings.

Scoring is primarily about saving lemmings quickly. Every lemming you save increases your score by 100 points. When you finish a level, you also get a time bonus equal to how many ticks remain on the level timer at the moment the level completes. Finishing faster therefore yields a higher score.

The builder cursor moves around the board using the arrow keys. When you press a tool key, if you still have that tool available in your tool inventory for the level and the square under the cursor is empty, the game places that tool onto the square and consumes one instance of the tool from your inventory. “Empty” here means that no actor at all (no floor brick, no tool, no hazard, no lemming, etc.) currently occupies that square. Tool placement is therefore both limited (by your inventory) and constrained (by what’s occupying squares at the moment you try to place it).

The builder cursor is not blocked by interior floor bricks and can move “through” objects; it’s best to think of it as a cursor hovering over the grid, not as a physical character in the maze. The placement rules still prevent you from placing a tool on top of something, but the cursor itself can sit anywhere inside the border bricks.

So how does a video game work?

Fundamentally, a video game is composed of a bunch of objects; in Lemmings, those objects include the builder cursor (the thing you move with the keyboard), a lemming factory that steadily produces lemmings, the lemmings themselves, solid floor bricks, hazards like roaming ice monsters and bonfires, and various placeable tools such as trampolines, nets, one-way doors, pheromones, springs, and the exit. Let's call these objects "actors," since each object is an actor in our video game. Each actor has its own x,y location in the 20×20 world, its own internal state (e.g., a lemming knows its location, what direction it's moving, whether it's walking/falling/bouncing/climbing, how far it has fallen, etc.), and its own special algorithms that control its actions in the game based on its own state and the state of the other objects in the world. In the case of the builder cursor, the algorithm that controls that actor object is the user's own brain and hand, and the keyboard. In the case of other actors (e.g., lemmings, ice monsters, factories, and hazards), each object has an internal autonomous algorithm and state that dictates how the object behaves in the world.

Once a level begins, gameplay is divided into ticks. A tick is a unit of time; in this project's framework it's roughly one frame of game logic (on the order of a few milliseconds, typically around 16ms, i.e., about 60 ticks per second). During a given tick, the game calls upon each object's behavioral algorithm and asks the object to perform its behavior. When asked to perform its behavior, each object's behavioral algorithm must decide what to do and then make a change to the object's state (e.g., move the object one square left/right/up/down, spawn a new lemming, change direction, kill or save a lemming, etc.), or change other objects' states (e.g., a bonfire might kill a lemming that moved onto it, an exit might save a lemming, a one-way door might force an actor to face a particular direction). Typically, the behavior exhibited by an object during a single tick is limited in order to ensure that the gameplay is smooth and that things don't move too quickly and confuse the user. For example, a lemming moves at most one square when it acts; an ice monster moves at most one square when it acts; a factory spawns at most one lemming at a time, and only at certain intervals.

After the current tick is over and all actors have had a chance to adjust their state (and possibly adjust other actors' states), our game framework animates the actors onto the screen in their new configuration. So, if a lemming changed its location from (10,5) to (11,5) (moved one square right), then our game framework will erase the graphic of the lemming from (10,5) on the screen and draw the lemming's graphic at (11,5) instead. Since this process (asking actors to do something, then drawing them in their updated positions) happens many times per second, the user will see smooth animation.

Then, the next tick occurs, and each object's algorithm is again allowed to do something, our framework displays the updated actors on-screen, etc.

Assuming the ticks are quick enough (a fraction of a second), and the actions performed by the objects are subtle enough (i.e., an actor doesn't move five squares away from where it was during the last tick, but instead changes position incrementally), when you display each of the

objects on the screen after each tick, it looks as if each object is performing a continuous series of fluid motions.

A video game can be broken into three different phases.

Initialization: The level (the 20×20 world) is initialized and prepared for play. This involves allocating one or more actors (which are C++ objects) and placing them in the world so that they will appear on the screen. In Lemmings, initialization includes loading the current level file (e.g., level00.txt), creating the floor bricks and hazards and tools specified in the file, creating the lemming factory, and placing the builder cursor into the world.

Game play: Game play is broken down into a bunch of ticks. During each tick, all of the actors in the world have a chance to do something, and perhaps die (lemmings can die; saved lemmings disappear from the world). During a tick, new actors may be added to the world (e.g., the factory may spawn a new lemming, the player may place a tool), and actors who die must be removed from the world and deleted.

Cleanup: The player has lost a life (but has more lives left), the player has completed the current level, or the player has lost all of their lives and the game is over. This phase frees all of the objects in the world (e.g., lemmings, the factory, hazards, floor bricks, tools, the exit, etc.) since the level attempt has ended. If the game is not over (i.e., the player has more lives), then the game proceeds back to Initialization, where the level is repopulated with new occupants and game play restarts at the current level file.

Here is what the main logic of a video game looks like, in pseudocode (we provide similar code in our provided GameController.cpp):

```
while (the player has lives left)
{
    Prompt_the_user_to_start_or_continue_playing();   // "Press Enter to begin/continue"
    Initialize_the_game_world();                      // You're going to write this

    while (the current level attempt is still active)
    {
        // each pass through this loop is a tick

        // You're going to write code to do the following
        Ask_all_actors_to_do_something();
        Delete_any_dead_actors_from_the_world();

        // we write this code to handle the animation for you
        Animate_all_of_the_actors_to_the_screen();
        Sleep_until_next_tick();  // the framework runs ticks at a steady rate
    }

    // the level attempt ended (either finished the level or lost a life)
    Cleanup_all_game_world_objects();                 // You're going to write this
    if (the player still has lives left)
        Prompt_the_Player_to_continue();
```

```
}

Tell_the_user_the_game_is_over();                    // the framework provides this
```

And here is what the Ask_all_actors_to_do_something() portion typically looks like:

```
Ask_all_actors_to_do_something()
{
    tell the build cursor to doSomething();

    for each actor currently on the level:
        if (the actor is still alive)
            tell the actor to doSomething();
}
```

You will typically use a container (an array, vector, or list) to hold pointers to each of your live actors. Each actor (a C++ object) has a doSomething() member function in which the actor decides what to do. In the doSomething() call, the actor updates its state appropriately each tick.

For example, here is some pseudocode showing what a (simplified) ice monster might decide to do each time it gets asked to do something:

```
class IceMonster : public SomeBaseClass
{
  public:
    void doSomething()
    {
        If a lemming is on my square, then
            Kill the lemming
        Else if it is time for me to move, then
            If the square in front of me is a solid brick, then
                Reverse my direction, but don't move this tick
            Else if the square in front of me has no solid ground beneath it, then
                Reverse my direction, but don't move this tick
            Else
                Move one square in my current direction
    }
};
```

And here's what the builder cursor's doSomething() member function looks like conceptually:

```
class Player : public …
{
  public:
    void doSomething()
```

```
    {
        Try to get user input (if any is available)

        If the user pressed an arrow key, then
            Move the cursor one square in that direction, but keep it inside the playable border

        If the user pressed a tool key (e.g., 'T', 'N', 'P', 'S', '<', '>') then
            If the cursor is on an empty square AND
               the player still has that tool available, then
                  Place that tool on the cursor's square and consume one instance of that tool
    }
};
```

What Do You Have to Do?

You must create a number of different classes to implement the Lemmings game. Your classes must work properly with our provided classes, and you must not modify our provided framework classes or our provided source files in any way to get your classes to work properly[1] (doing so will result in a score of zero on the entire project). The core requirement is that you implement the StudentWorld class, which is responsible for keeping track of the game world (including the 20×20 grid and everything in it) and all of the actors/objects (lemmings, the lemming factory, floor bricks, hazards, tools, the exit, and the builder cursor) that are inside the world. In addition, you must implement the actor classes that model the behaviors of the objects in the world, including the builder cursor, lemmings, the lemming factory, floor bricks, ice monsters, bonfires, the exit, and the various tools (trampolines, nets, one-way doors, pheromones, and springs). You may also create any additional base classes (for example, a base "Actor" class or a base "MovingActor" class) that help you implement the game cleanly.

You Have to Create the StudentWorld Class

Your StudentWorld class is responsible for orchestrating virtually all game play. It keeps track of the game world (the 20×20 grid and all of its inhabitants such as lemmings, the factory, ice monsters, bonfires, floor bricks, tools, and the exit). It is responsible for initializing the world at the start of a level attempt, asking all the actors to do something during each tick, destroying an actor when it disappears (e.g., a lemming dies or is saved), and destroying all of the actors in the world when the user loses a life or completes the level.

Your StudentWorld class must be derived from our GameWorld class (found in GameWorld.h) and must implement at least these three methods (which are defined as pure virtual in our GameWorld class):

```
virtual int init() = 0;
virtual int move() = 0;
virtual void cleanUp() = 0;
```

[1] With one exception: At the top of main.cpp, you may change the string literal used to initialize the assetDirectory variable..

The code that you write must never call any of these three functions. Instead, our provided game framework will call these functions for you. So, you have to implement them correctly, but you won't ever call them yourself in your code.

init() Overview

When a new level starts (e.g., at the start of the game, or when the player completes a level and advances to the next level), our game framework will call the init() method that you defined in your StudentWorld class. You don't call this function; instead, our provided framework code calls it for you.

The init() method is responsible for loading the current level's 20×20 layout from a data file (described later), and constructing a representation of the current level in your StudentWorld object, using one or more data structures that you come up with.

The init() method is automatically called by our provided code either when the game first starts, when the player completes the current level and advances to a new level that needs to be loaded/initialized, or when the user loses a life (but has more lives left) and the game is ready to restart the current level from scratch.

When the player has finished the level loaded from level00.txt, the next level data file to load is level01.txt; after level01.txt, level02.txt; etc. If there is no level data file with the next number, the init() method must return GWSTATUS_PLAYER_WON. If the next level file exists but is not in the proper format for a level data file, the init() method must return GWSTATUS_LEVEL_ERROR. Otherwise, the init() method must return GWSTATUS_CONTINUE_GAME.

move() Overview

Once a new level has been loaded/initialized with a call to init(), our game framework will repeatedly call the StudentWorld's move() method at a steady rate, once per tick. Each time the move() method is called, it must run a single tick of the game. This means that it is responsible for asking each of the actors in the world to try to do something: e.g., move themselves and/or perform their specified behavior, spawn new lemmings, kill lemmings, save lemmings, and place tools. Finally, this method is responsible for disposing of (i.e., deleting) actors (e.g., a lemming that died in a bonfire, or a lemming that was saved by reaching the exit) that need to disappear during a given tick. The move() method will automatically be called once during each tick of the game by our provided framework. You will never call move() yourself.

cleanUp() Overview

The cleanUp() method is called by our framework when the player completes the current level or loses a life. In Lemmings, “loses a life” means the level attempt ended due to giving up, running out of time before saving enough lemmings, or losing too many lemmings to death. The cleanUp() method is responsible for freeing all actors (e.g., all lemming objects, all floor bricks, all tools, the factory, hazards, the exit, etc.) that are currently in the world. This includes all actors created during init() or introduced during subsequent game play (e.g., lemmings spawned by the factory, tools placed by the player) that have not yet been removed from the world.

You may add as many other public/private member functions or private data members to your StudentWorld class as you like (in addition to the above three member functions, which you must implement).

Your StudentWorld class must be derived from our GameWorld class. Our GameWorld class provides the following methods for your use:

```
int getLevel() const;
int getLives() const;
void decLives();
int getScore() const;
void increaseScore(int howMuch);
void setGameStatText(std::string text);
std::string assetPath() const;
bool getKey(int& value);
void playSound(int soundID);
```

getLevel() can be used to determine the current level number (0 for level00.txt, 1 for level01.txt, etc.).

getLives() can be used to determine how many lives the player has left. The game begins with three lives.

decLives() reduces the number of lives by one.

getScore() can be used to determine the player’s current score.

increaseScore() is used by your StudentWorld object (or your other classes) to increase the user’s score when lemmings are saved and when a level is completed (to give the player their remaining time bonus). When your code calls this method, you must specify how many points the user gets. This means that the game score is controlled by our GameWorld object; you must not maintain your own separate “score” data member in your own classes.

The setGameStatText() method is used to specify what text is displayed at the top of the game screen.

assetPath() returns the name of the directory that contains the game assets (image, sound, and level data files).

getKey() can be used to determine if the user has hit a key on the keyboard during the current tick. This method returns true if the user hit a key during the current tick, and false otherwise. The argument to this method is a variable that will be set to the key that was pressed by the user (if any key was pressed). When getKey() returns true, the argument will be set either to one of the special key codes defined in GameConstants.h for arrow keys (e.g., KEY_PRESS_LEFT, KEY_PRESS_RIGHT, KEY_PRESS_UP, KEY_PRESS_DOWN), or to the encoding of a typed character used for tool placement (e.g., 'T', 'N', 'P', 'S', '<', '>', etc.).

playSound() can be used to play a sound effect when an important event happens during the game (e.g., a lemming is saved, a lemming dies, or the level is finished). You can find constants (e.g., SOUND_LEMMING_SAVED) that describe what sound to make in GameConstants.h. Here's how this method might be used:

```
// A lemming died, so make a dying sound
playSound(SOUND_LEMMING_DIE);
```

init() Details

Your StudentWorld's init() member function must initialize the data structures used to keep track of your game's world, load the current level details from a level data file, allocate and insert the world's initial actors into the game world (floor bricks, the lemming factory, ice monsters, bonfires, any pre-placed tools, and the exit), and allocate the builder cursor (the "player" object) and place it into the world. The builder cursor is not specified in the level file; it begins each level at the center of the board, at coordinate (VIEW_WIDTH / 2, VIEW_HEIGHT / 2), i.e., (10,10) in a 20×20 world.

In addition, init() must reset all per-level variables that control victory and defeat. Each level attempt has a fixed time limit measured in ticks; the level timer starts at 2000 ticks for each attempt and decreases by 1 each tick. Each level spawns exactly 10 lemmings total, and the lemming factory must not spawn more than this maximum. Your world must track how many lemmings have been spawned so far, how many have been saved so far, and how many have died so far during the current attempt.

To load the details of the current level from a level data file, you can use the Level class (described later) that we provide, which can be found in the provided Level.h header file. Here's a brief example showing some uses of the Level class interface to load a level data file:

```
#include "Level.h"   // you must include this file to use our Level class
#include "Coord.h"   // for the Coord struct
```

```
int StudentWorld::someFunctionYouWriteToLoadALevel()
{
    std::string curLevel = "level03.txt";
    Level lev(assetPath());
    Level::LoadResult result = lev.loadLevel(curLevel);

    if (result == Level::load_fail_file_not_found ||
        result == Level::load_fail_bad_format)
        return -1;   // something bad happened!

    // otherwise the load was successful and you can access the
    // contents of the level

    Coord p(0, 5);
    Level::MazeEntry item = lev.getContentsOf(p);
    if (item == Level::lemming_factory)
        std::cout << "A lemming factory should be placed at 0,5\n";

    Coord q(10, 7);
    item = lev.getContentsOf(q);
    if (item == Level::floor)
        std::cout << "There should be a floor brick at 10,7\n";

    ...
}
```

Notice that coordinates are represented using a Coord struct (defined in Coord.h) with fields x and y. In this project, x is the column (0 to 19) and y is the row (0 to 19). The bottom-left of the world is (0,0), and y increases upward. In the level file itself, the first line of the 20×20 grid corresponds to the top row (y=19), and the last of the 20 lines of the grid corresponds to the bottom row (y=0).

Once you load a level's layout and details using our Level class, your init() method must then construct a representation of your world and store this in your StudentWorld object. **It is required that you keep track of all of the actors (e.g., lemmings, floor bricks, ice monsters, bonfires, tools, the exit, etc.) in a single STL collection like a list or vector, using a container of pointers to actors. Your StudentWorld object *may optionally* keep a separate pointer to the builder cursor object rather than keeping a pointer to that object in the container with the other actor pointers; the builder cursor is the only actor allowed to not be stored in the single actor container.**

You must not call init() yourself. Instead, this method will be called by our framework code when it's time for a new level attempt to start.

Level Data Files

Each level data file is named levelNN.txt, where NN is a two-digit number starting at 00. Each file contains 20 lines of 20 characters each, describing the 20×20 world grid, followed by one additional line that describes the tools available to the player for that level attempt.

Within the 20×20 grid portion of the file, each character represents what occupies that square at the start of the level:

- `' '` (space): Empty square
- `'@'`: Solid floor brick (or wall) that can be walked on and blocks movement
- `'L'`: Lemming factory
- `'I'`: Ice monster
- `'B'`: Bonfire
- `'T'`: Trampoline already present in the world
- `'N'`: Net already present in the world
- `'<'`: Left-facing one-way door already present in the world
- `'>'`: Right-facing one-way door already present in the world
- `'P'`: Pheromone already present in the world
- `'S'`: Spring already present in the world
- `'X'`: Exit

Any other character in the 20×20 grid makes the file invalid and causes `init()` to return `GWSTATUS_LEVEL_ERROR`.

In addition to validating the character set and line lengths, the level loader enforces structural requirements. The level must include exactly one lemming factory and at least one exit. The level must also satisfy the boundary requirement: the outer edges of the world must be floor bricks, so that lemmings and other actors cannot leave the playable area. If these conditions are not met, the level file is considered malformed.

After the 20 lines of the grid, the file must contain one additional line: the tools line. The tools line is a string that represents the player's tool inventory for the level. Each character in that string is one tool instance the player may place. If a tool character appears multiple times, the player may place that tool multiple times. The valid tool characters are the same characters used for tool placement during gameplay:

- 'T' for trampoline
- 'N' for net
- 'P' for pheromone
- 'S' for spring
- '<' for a left-facing one-way door, and
- '>' for a right-facing one-way door.

For example, the tools line

TTNS<PPP

indicates that the player will be able to place two trampolines (T), one net (N), one spring (S), one left-facing door (<) and three pheromones (P).

move() Details

The move() method must perform one tick of simulation and must return a status code indicating whether the level attempt should continue, whether the player lost a life, or whether the level was finished.

At the start of each tick, the move() method must decrement the level's timer by one tick. The level timer begins at 2000 ticks for each attempt and counts down toward zero. This timer is used both for the loss condition (if it reaches zero before enough lemmings have been saved) and for the time bonus when the level is completed (the bonus is the number of ticks remaining at the moment the level completes).

During each tick, move() must ask every actor (including the builder cursor and actors like lemmings, ice monsters, etc.) in the world to do something, by calling each actor's doSomething() method if the actor is alive. The builder cursor is responsible for moving itself or for placing a tool when the user presses the appropriate key. The other actors are responsible for autonomous simulation: factories spawn lemmings, lemmings walk/fall/bounce/climb, hazards kill lemmings, and exits save lemmings.

It is possible that during its turn in the current tick, one actor (e.g., an Ice Monster) may cause another actor (e.g., a Lemming) to die before it had its chance to act during the current tick. In that case, the victim actor must NOT do anything during the current tick (since it's dead).

To help you with testing, if you press the f key during the course of the game, our game controller will stop calling move() every tick; it will call move() only when you hit a key (except the r key). Freezing the activity this way gives you time to examine the screen, and stepping one move at a time when you're ready helps you see if your actors are moving properly. To resume regular game play, press the r key.

After all actors have had a chance to act during the tick, move() must delete any actors that are no longer alive (for example, lemmings that died or were saved). This includes removing their pointers from the StudentWorld's actor container, and deleting the objects to avoid memory leaks.

After updating and cleanup for the tick, move() must check whether the level attempt should end. Each level spawns exactly 10 lemmings. To complete the level, the player must save at least half of them to complete the level, which means saving 5 or more lemmings.

The level attempt ends immediately (and the player loses a life) if more than half of the total number of lemmings to be spawned have died (from falling or encounters with ice monsters). With 10 lemmings, this means the attempt fails as soon as the dead-lemming count becomes 6 or more. When this happens, move() must call decLives() and must return GWSTATUS_PLAYER_DIED.

The level attempt also ends when the timer reaches zero. If the timer reaches zero and the player has not saved enough lemmings, the player loses a life: move() must call decLives() and return GWSTATUS_PLAYER_DIED. If the timer reaches zero but the player has already saved enough lemmings, the level is considered finished; the time bonus at that point is zero (since no ticks remain), and move() must return GWSTATUS_FINISHED_LEVEL (after playing the appropriate completion sound, SOUND_FINISHED_LEVEL).

If the timer has not reached zero, the level is considered finished when the world has spawned all lemmings for the level and there are no lemmings left alive in the world, and the number of saved lemmings is at least the required minimum of 5. When the level finishes in this way, move() must increase the player's score by the number of ticks remaining on the timer, play the finished-level sound, SOUND_FINISHED_LEVEL, and return GWSTATUS_FINISHED_LEVEL.

If none of the ending conditions occur, move() must return GWSTATUS_CONTINUE_GAME to indicate that the next tick should proceed normally.

The move() method must update the status text on the top of the screen during every tick by calling the setGameStatText() method that we provide in our GameWorld class. The status line must include the current score, level number, lives remaining, number of saved lemmings, the remaining tools inventory, and the time left on the level timer.

Updating the Display Text

Your move() method must update the game statistics at the top of the screen during every tick by calling setGameStatText(). The format must match the following pattern:

```
Score: #####  Level: ##  Lives: ##  Saved: ##  Tools: <tools-string-or-None>  Time left: ####
```

Each field's label is followed by a colon and one space. Each statistic is separated from the previous statistic by exactly two spaces. The Score field must be displayed as exactly 5 digits with leading zeros. The Level field must be displayed as exactly 2 digits with leading zeros. The Lives field must be displayed as exactly 2 digits with leading zeros. The Saved field must be displayed as exactly 2 digits with leading zeros. The Time left field must be displayed as exactly 4 digits with leading zeros. The Tools field is not a fixed-width number; it must display the remaining tools in the player's inventory (with the tool characters in an order of your choosing), or the word "None" if the player has no remaining tools.

Scoring and Lives

The player begins the game with three lives. A level attempt ends and the player loses one life if the player gives up on the attempt (by pressing the G key, for instance if they know they cannot possibly win the level given their current tool placement), if the level timer reaches zero before the player has saved enough lemmings, or if more than half the lemmings die. If the player still has lives left after losing a life, the framework will restart the current level from scratch by calling cleanUp() and then init() again. If the player has no lives left, the game ends.

Scoring is primarily about saving lemmings quickly. Each lemming saved (when it reaches an exit) increases the score by 100 points. When the level finishes, the player also receives a time bonus equal to the number of ticks remaining on the level timer at the moment the level completes, which encourages faster solutions.

The Player (Builder Cursor) and Tool Placement

The builder cursor is moved directly by the user via the arrow keys. The cursor is not blocked by floor bricks and can move “through” objects; it is best to think of it as a cursor hovering over the grid, not as a physical character that collides with terrain. Even though it can hover over anything, it is constrained to remain inside the border of the level and must not move to the positions of the outermost floor brick squares.

When the user presses a tool key (e.g., 'N'), if the player still has that tool (e.g., a net) available in the tool inventory for the level and the square under the cursor is empty, the game places that tool onto the square and consumes one instance of the tool from the inventory. “Empty” here means that no actor at all (no floor brick, no tool, no hazard, no lemming, etc.) currently occupies that square; the cursor itself does not count as occupying the square for this purpose. If the square is not empty, or if the player has no remaining instance of that tool in the inventory, then nothing happens.

To place a tool in the game, the player/human will press one of six different keys:

- The right-facing one-way door is placed with the '>' key.
- The left-facing one-way door is placed with the '<' key.
- A trampoline is placed with the 'T' key (either 'T' or 't' is accepted).
- A net is placed with the 'N' key (either 'N' or 'n' is accepted).
- A pheromone is placed with the 'P' key (either 'P' or 'p' is accepted).
- A spring is placed with the 'S' key (either 'S' or 's' is accepted).

Each placement consumes one matching character from the remaining tools inventory.

Lemming Simulation Rules

Lemmings are not directly controlled by the user. They are autonomous actors with a simple state machine that determines whether they are walking, falling, bouncing, or climbing.

Lemmings generally travel horizontally (left or right) while walking. If a lemming attempts to move forward into a solid floor brick, it reverses direction and does not move forward that tick. When walking, a lemming also checks whether it is currently on a climbable tool (a net). If it is, it transitions into climbing instead of continuing to walk.

If a lemming is walking and steps forward into a square that does not have a floor brick directly beneath it, the lemming begins to fall. Falling lemmings move downward one square when they act, continuing until they reach a square where they can no longer move downward because a floor brick is beneath them. The world tracks how far a lemming has fallen; if the lemming lands after falling more than five squares, it dies. If it lands after falling five squares or fewer, it survives and transitions back to walking.

Bouncing lemmings are created when a falling lemming is affected by certain tools. Trampolines interact with falling lemmings and cause them to bounce upward, reducing the danger of long falls. Springs also cause a lemming to bounce upward with a large bounce (even by just walking onto one). While bouncing, the lemming moves upward for the bounce, then moves horizontally once at the apex, and then transitions into falling again. If a lemming overlaps a net while falling or bouncing, it may transition into climbing.

Climbing lemmings move upward one square when they act, but only while they remain on climbable squares (nets). If a climbing lemming reaches a square that is not a net, it transitions back to walking.

Pheromones influence lemming direction. If a pheromone exists five or fewer squares directly to the left or right of a lemming on the same row, the lemming turns to face the closest such pheromone (turning left if the pheromone is to the left, turning right if the pheromone is to the right). If multiple pheromones qualify, any one of the closest ones is used to determine the direction.

Hazards and Saving Lemmings

Bonfires and ice monsters are hazards that kill lemmings that overlap them. When a lemming dies, it is removed from the world, the dead-lemming count increases by one, and the appropriate sound effect is played.

The exit is the goal. When a lemming overlaps the exit, that lemming is saved: it is removed from the world, the saved-lemming count increases by one, the player’s score increases by 100 points, and the appropriate sound is played. Saved lemmings do not count as dead for purposes of the “too many deaths” failure condition.

cleanUp() Details

When your cleanUp() method is called by our game framework, it means that the player lost a life or has completed the current level. In this case, every actor in the entire world, including the building cursor object, must be deleted and removed from the StudentWorld’s container of

active objects, resulting in an empty world. If the user has more lives left, our provided code will subsequently call init() to reload and repopulate the world and the level attempt will then continue from scratch with a brand new set of actors.

You must not call cleanUp() yourself when the level attempt ends. Instead, this method will be called by our code.

A note on giving up: the framework supports a give-up key. If the user presses 'G' (or 'g') during gameplay, our framework (not your code) will immediately treat the level attempt as failed, decrementing the player's lives, calling cleanUp(), and calling init() to restart the current level attempt if any lives remain.

IMPORTANT NOTE: The StudentWorld destructor will be called by our game framework when the game is over. If the game ends prematurely because the user pressed the q key, cleanUp() will NOT have been called by our framework, so your destructor should call it to make sure the game shuts down cleanly. In normal gameplay, the Player may have no more lives or may finish the last level, resulting in cleanUp() being called as for any level ending; a little later, the StudentWorld destructor is called, which would call cleanUp() again. MAKE SURE two consecutive calls to cleanUp() won't do anything undefined. For example, if cleanUp() deletes an object and leaves a dangling pointer, it could be disastrous if the second call to cleanUp() tries to use that pointer in a delete expression.

The Level Class and Level Data File

As mentioned, every level of Lemmings has a different 20×20 "world" layout. The layout for each level is stored in a text data file, with the file level00.txt holding the details for the first level, level01.txt holding the details for the second level, etc. Several sample level files are included in the Assets folder we provide.

Here's an example level data file. You are welcome to modify our level data files to create wacky new levels or levels to make your testing easier, or to add your own new additional level data files.

```
@@@@@@@@@@@@@@@@@@@@
@                  @
@                  @
@        @@@@      @
@                  @
@                  @
@                  @
@                  @
@         @        @
@                  @
@       X          @
@     @@@          @
@                  @
@             @@@@@@@
@ I                @
@@@@@@@@@@         @
@        B         @
@        @@@@@     @
@                  @
@  L              B@
@@@@@@@@@@@@@@@@@@@@
SSSPPNN<
```

As you can see, the data file contains a 20×20 grid of different characters that represent the different actors in the level, followed by one extra line (after the 20 grid lines) that specifies the tools available to the player for that level attempt (three springs, two pheromones, two nets and a left-facing one-way door).

Valid characters for your level data file's 20×20 grid are:

- The @ character represents a floor brick (terrain). Floor bricks are solid; they block movement into their square. The level's boundary must be surrounded by floor bricks.
- The L character specifies the location of the lemming factory. Every valid level must contain one lemming factory location in the 20×20 grid.
- The X character specifies the location of the level exit. Every valid level must contain at least one exit location in the 20×20 grid (our provided levels use exactly one).
- The I character represents an ice monster (a roaming hazard).
- The B character represents a bonfire (a stationary hazard).
- The T character represents a trampoline.
- The N character represents a net (a climbable "ladder" tile).
- The < character represents a left-facing one-way door (a direction-forcing tile that makes actors face/move left).
- The > character represents a right-facing one-way door (a direction-forcing tile that makes actors face/move right).

- The P character represents a pheromone (an attractor tile that can influence lemming direction).
- The S character represents a spring (a launcher tile).
- All space characters represent empty air squares.

Immediately after the 20 grid lines, the file must contain one additional line: the tools line. This tools line is a string of characters that specifies the player's inventory of placeable tools for that level attempt. Each character in the tools line represents one available use of that tool. For example, a tools line of:

NTT>

means the player starts the attempt with one net (N), two trampolines (T and T), and one right-facing one-way door (>).

Only the following characters are allowed in the tools line:

T (trampoline), N (net), P (pheromone), S (spring), < (left one-way door), and > (right one-way door).

If a tool character appears multiple times, the player may place that tool multiple times during the level (once per occurrence), as long as each placement follows the placement rules described later.

The Level Class

We have graciously 🙂 decided to provide you with a class that can load level data files for you. The class is called Level and may be found in our provided Level.h file. Here's an example showing some uses of the Level class interface to load a level data file:

```
#include "Level.h" // required to use our provided class
void StudentWorld::someFunc()
{
  Level lev(assetPath()); // note: GameWorld provides assetPath()
  Level::LoadResult result = lev.loadLevel("level00.txt");
  if (result == Level::load_fail_file_not_found)
    cerr << lev.getErrorMessage() << endl;
  else if (result == Level::load_fail_bad_format)
    cerr << lev.getErrorMessage() << endl;
  else if (result == Level::load_success)
  {
    cerr << "Successfully loaded level\n";
```

```
Coord c(5, 10);  // x=5, y=10
Level::MazeEntry me = lev.getContentsOf(c);

switch (me)
{
    case Level::empty:
        cout << "5,10 is empty\n";
        break;
    case Level::floor:
        cout << "5,10 holds a floor brick\n";
        break;
    case Level::lemming_factory:
        cout << "5,10 holds the lemming factory\n";
        break;
    case Level::ice_monster:
        cout << "5,10 starts with an ice monster\n";
        break;
    case Level::bonfire:
        cout << "5,10 starts with a bonfire\n";
        break;
    case Level::trampoline:
        cout << "5,10 starts with a trampoline\n";
        break;
    case Level::net:
        cout << "5,10 starts with a net\n";
        break;
    case Level::left_one_way_door:
        cout << "5,10 starts with a left one-way door\n";
        break;
    case Level::right_one_way_door:
        cout << "5,10 starts with a right one-way door\n";
        break;
    case Level::pheromone:
        cout << "5,10 starts with a pheromone\n";
        break;
    case Level::spring:
        cout << "5,10 starts with a spring\n";
        break;
    case Level::lemming_exit:
        cout << "5,10 is where the exit is\n";
        break;
}
```

```
    string tools = lev.getTools();
    cout << "Tools line is: " << tools << "\n";
  }
}
```

Hint: You will presumably want to use our Level class when loading the current level specification in your StudentWorld's init() method.

You Have to Create the Classes for All Actors

The Lemmings game has a number of different game objects, including:

• The player's builder cursor
• Lemmings
• A lemming factory (that steadily produces lemmings over time)
• Floor bricks (terrain)
• Ice monsters (roaming hazards)
• Bonfires (stationary hazards)
• Trampolines (a tool tile that launches lemmings)
• Nets (a tool tile that allows climbing)
• One-way doors (left- and right-facing)
• Pheromones (a tool tile that attracts lemmings horizontally)
• Springs (a tool tile that launches lemmings)
• Exits (lemmings use this to exit the level)

Each of these game objects can occupy squares in the 20×20 world and interact with other objects during the simulation.

Now of course, many of your game objects will share things in common. Every one of the objects in the game (lemmings, hazards, tools, floor bricks, etc.) has a location on the grid. Many game objects need an opportunity to act during each tick of the game (lemmings move; the factory spawns; ice monsters patrol). Many actors activate when a lemming steps on them and performs some action upon that lemming (bouncing the lemming, changing the lemming's direction).

It is therefore your job to determine the commonalities between your different game objects and make sure to factor out common behaviors and traits into appropriate base classes, rather than duplicate these items across your derived classes. This is one of the tenets of object-oriented programming.

Your grade on this project will depend upon your ability to intelligently create a set of classes that follow good object-oriented design principles. Your classes should not duplicate code or data members. If you find yourself writing the same (or largely similar) code across multiple

classes, then this is an indication that you should define a common base class and migrate this common functionality/data to the base class. Duplication of code is a code smell that often leads to bugs, inconsistencies, and code bloat.

Hint: When you notice this specification repeating similar requirements in the following sections (e.g., in the Trampoline and Spring sections, or in the Bonfire and Ice Monster sections), you must identify common behaviors and move these into proper base classes. NEVER duplicate behaviors across classes that can be moved into a base class!

Deriving From GraphObject

You MUST derive all of your game objects directly or indirectly from a base class that we provide called GraphObject, e.g.:

```
class Actor : public GraphObject
{
public:
...
};
class Lemming : public Actor
{
public:
...
};
```

GraphObject is a class that we have defined that helps hide the ugly logic required to graphically display your actors on the screen. If you don't derive a class from our GraphObject base class, then you won't see any object of that class displayed on the screen!

GraphObject uses a Coord struct (defined in Coord.h) to represent grid positions. A Coord has two integers that represent a position on the screen: x and y.

The GraphObject class provides the following methods that you may use:

- GraphObject(int imageID, Coord startCoord, int dir = right);
- void setVisible(bool shouldIDisplay);
- Coord getCoord() const;
- void moveTo(Coord coord);
- void moveTo(int direction);
- int getDirection() const;
- void setDirection(int d);
- Coord getTargetCoord(int dir) const;
- Coord getTargetCoord(Coord base, int dir) const;

You may use these member functions in your derived classes, but you must not use any other member functions found inside of GraphObject in your other classes (even if they are public in our class). You must also be careful about redefining GraphObject behavior: for example, GraphObject already manages an object's position; your derived classes must not keep redundant x/y member variables.

GraphObject(int imageID, Coord startCoord, int dir) is the constructor for a new GraphObject. When you construct a new GraphObject, you must specify:

1. An image ID that indicates how the GraphObject should be displayed on screen (e.g., as a lemming, an ice monster, a floor brick, etc.).
2. The initial grid position (startCoord.x, startCoord.y).
3. A direction, when direction matters for the object.

The coordinate's x value may range from 0 to VIEW_WIDTH-1 inclusive (0–19), and the y value may range from 0 to VIEW_HEIGHT-1 inclusive (0–19).

In this project, GraphObject directions may be any of the following:

GraphObject::right (0 degrees)
GraphObject::left (180 degrees)
GraphObject::up (90 degrees)
GraphObject::down (270 degrees)
GraphObject::none (-1)

For example, when you call moveTo(int direction), you pass one of these direction constants and the object moves one square in that direction. When you call getTargetCoord(...), it returns the adjacent coordinate one square away in the given direction.

One of the following IDs, found in GameConstants.h, must be passed in for the imageID value when constructing each actor to determine what sprite we display on the screen:

IID_PLAYER
IID_LEMMING_FACTORY
IID_LEMMING
IID_FLOOR
IID_ICE_MONSTER
IID_BONFIRE
IID_TRAMPOLINE
IID_NET
IID_ONE_WAY_DOOR
IID_EXIT
IID_PHEROMONE
IID_SPRING

Getting Input From the User

Since Lemmings is a real-time game, you can't use getline or cin to get user input during a tick. Instead, you will use a function called getKey() that we provide in our GameWorld class (from which your StudentWorld class is derived). This function rapidly checks to see if the user has hit a key. If so, it returns true and sets the int variable you pass to the code for the key; otherwise, it returns false immediately.

This function could be used as follows:

```
void Player::doSomething()
{
  int ch;
  if (getWorld()->getKey(ch))
  {
    switch (ch)
    {
      case KEY_PRESS_LEFT:
        ... move builder cursor left ...
        break;
      case KEY_PRESS_RIGHT:
        ... move builder cursor right ...
        break;
      case 'T':
      case 't':
        ... place a trampoline tool ...
        break;
        // etc.
    }
  }
}
```

In this project:

1. The arrow keys move the builder cursor (the "player" object) around the screen.
2. Tool placement keys are ordinary character keys: 'T', 'N', 'P', 'S', '<', and '>'.
3. The 'G' key (or 'g') causes the player to give up on the current level attempt. This is handled by our framework: giving up immediately costs one life and restarts the current level from scratch. You do not need to implement the 'G' key behavior inside your Player class; the framework handles it before calling StudentWorld::move().

The Player

Here are the requirements you must meet when implementing the player's builder cursor class (often named Player, even though it is a cursor and not a maze-walking avatar).

What the Builder Cursor Must Do When It Is Created

When it is first created:

1. The builder cursor object must have an image ID of IID_PLAYER.
2. The builder cursor starts at the center of the grid (x=VIEW_WIDTH/2, y=VIEW_HEIGHT/2).
3. The builder cursor starts facing right (its direction will never change).

In addition to any other initialization that you decide to do in your builder cursor class, it must be visible.

What the Builder Cursor Must Do During a Tick

The builder cursor must be given an opportunity to do something during every tick (in its doSomething() method). When given an opportunity to do something, the builder cursor must do the following:

1. The builder cursor must check to see if the user pressed a key using getKey(). If the user pressed a key:
   a. If the user pressed a directional key (up, down, left, right), the builder cursor moves one square in that direction, subject to the boundary constraint described below.
   b. If the user pressed a tool key ('T', 'N', 'P', 'S', '<', '>'), then the builder cursor attempts to place the corresponding tool on the square currently under the cursor, subject to the placement rules described below.

The builder cursor is not blocked by terrain or by any actors. It is best to think of it as a cursor hovering over the grid, not as a physical character in the grid. This means the builder cursor may move "through" floor bricks, hazards, tools, and so on.

However, the builder cursor is constrained to remain inside the border of the world. It may not move onto the outermost boundary squares. Concretely, it may move only within the interior region $1 \le x \le$ VIEW_WIDTH-2 and $1 \le y \le$ VIEW_HEIGHT-2.

Tool Placement Rules

When the user presses a tool key, the game attempts to place that tool onto the square under the cursor. The game places the tool only if BOTH of the following are true:

1. The player still has at least one instance of that tool available in the current level's tool inventory (as specified by the tools line from the level file, minus any tools already placed this attempt).
2. The square under the cursor is empty.

"Empty" here has a strict meaning: the square must contain no actor at all (no floor brick, no tool, no hazard, no exit, no factory, and no lemming). If any actor occupies that square at the moment the tool key is pressed, then the tool is not placed and the player does not consume any tool from their inventory.

If the tool is successfully placed, then:

1. A new tool actor is added to the world at that square.
2. One instance of that tool is consumed from the tool inventory for the level attempt.

The player object itself does not occupy the square for placement purposes; only "real actors" in the world block placement. The player object is not launchable (by a trampoline or spring), does not block movement, and is not climbable.

Floor Brick

You must create a class to represent a floor brick. Floor bricks are solid terrain tiles that make up floors and walls. Here are the requirements you must meet when implementing the FloorBrick class.

What a Floor Brick Must Do When It Is Created

When it is first created:

1. A floor brick object must have an image ID of IID_FLOOR.
2. A floor brick starts at the location specified by the current level's data file.
3. A floor brick starts facing right.

What a Floor Brick Must Do During a Tick

A floor brick must be given an opportunity to do something during every tick (in its doSomething() method). When given an opportunity to do something, the floor brick must do nothing. After all, it's just a brick!

What a Floor Brick Must Do Regarding Movement Blocking

Floor bricks are solid. If asked if they obstruct movement, they must indicate that they do. Any actor whose movement rules say "do not move into solid squares" can then treat a floor brick as

an obstruction that blocks entry into the floor brick's square. Floor bricks are not launchable from a trampoline or spring, nor are they climbable.

The Lemming Factory

You must create a class to represent the lemming factory. The lemming factory steadily produces lemmings over time until the level's lemming quota has been spawned. Here are the requirements you must meet when implementing the LemmingFactory class.

What the Lemming Factory Must Do When It Is Created

When it is first created:

1. A lemming factory object must have an image ID of IID_LEMMING_FACTORY.
2. The lemming factory starts at the location specified by the current level's data file.
3. The lemming factory starts facing right.

What the Lemming Factory Must Do During a Tick

Each time a lemming factory is asked to do something (during a tick):

1. The factory keeps track of how many ticks have passed since it last spawned a lemming.
2. Every 100 ticks, if the factory has not yet spawned the maximum number of lemmings for the level, the factory attempts to spawn one new lemming. The first lemming is thus spawned at tick 100, not at tick zero.
3. The lemming is spawned on the same square as the factory. It's your choice whether the lemming is first asked to do something during the current tick or the next tick.
4. The newly spawned lemming's initial direction is right.
5. The factory stops spawning new lemmings after it has spawned 10 lemmings.

The lemming factory is not solid; if asked if it blocks movement it must indicate that it does not. This also means lemmings and ice monsters may overlap the factory square. The lemming factory is not launchable from a trampoline or spring, nor is it climbable.

Lemming

You must create a class to represent a lemming. Lemmings are the autonomous creatures you are trying to guide to the exit. The player does not directly control lemmings; instead, lemmings move according to simple rules that depend on their current movement state (walking, falling, bouncing, or climbing), plus influences from placed tools like pheromones, nets, and bounce platforms.

What a Lemming Must Do When It Is Created

When it is first created:

1. A lemming object must have an image ID of IID_LEMMING.
2. A lemming must have its starting Coord specified when it is constructed. Normally, this is the lemming factory's Coord, since the factory spawns lemmings onto its own square.
3. A lemming must have an initial direction of right.
4. A lemming starts in the walking movement state.
5. A lemming starts with its internal movement-timing counters reset (described below).

A lemming is not solid; if asked if it blocks movement by another actor, it must indicate that it does not block movement. Multiple lemmings and ice monsters can occupy the same square, and lemmings can overlap with non-solid tools and hazards (e.g., pheromones, nets, bonfires, exits, one-way doors). A lemming *is* launchable from a trampoline or spring. A lemming is not climbable.

What a Lemming Must Do During a Tick

The lemming must check to see if it is currently alive. If not, then its doSomething() method must return immediately.

Otherwise it proceeds as follows:

Step 1: Checking if it should move

Although a lemming is given an opportunity to do something every tick (through its doSomething logic), it does not actually run its movement rules every tick. Instead, lemming movement is intentionally slowed down.

If in the walking state, a lemming will try to walk once every 4 ticks. It is idle the other 3 ticks and immediately returns.

If in any other state (e.g., falling, bouncing, crawling), a lemming will move once every 2 ticks. It is idle every other tick and immediately returns.

If the lemming is in an idle state, its doSomething() method immediately returns without running. Otherwise, if a lemming is not in an idle state, it must apply the following behaviors in the order described:

Step 2: Process Pheromone Attraction

The lemming must check for pheromones that can attract it:

1. A pheromone attracts a lemming only if the pheromone is on the same row as the lemming (same y coordinate).
2. A pheromone attracts a lemming only if it is 5 or fewer squares horizontally of the lemming ($|\Delta x| \leq 5$).
3. A pheromone attracts a lemming only if it is not on the same square as the lemming (Δx cannot be 0).
4. If at least one pheromone satisfies these conditions, the lemming must turn to face the closest such pheromone (left or right). If multiple pheromones are tied for closest distance, it's your choice which of the closest pheromones the lemming ends up facing.

Step 3: Process Movement (Walking, Falling, Climbing, Bouncing)

Determine which state the lemming is in, and run through those steps.

Walking State

If the lemming is in the walking state and is not currently idle, then it must do the following:

1. First, if the lemming's current square contains a climbable actor (i.e., a net), then the lemming must immediately transition to the climbing state and do nothing more during this tick. (It does not move on the same update that it switches into climbing.)
2. Otherwise, the lemming considers the adjacent square one step in the direction it is currently facing. Let's call this square "next".
3. If next contains a solid obstruction (specifically, a floor brick), then the lemming must not move. Instead, it must reverse direction (left becomes right, right becomes left), and then do nothing more during this tick.
4. Otherwise, the lemming considers the square directly below next (one square down from next). Let's call this square "belowNext".
5. If belowNext contains a solid floor brick, then the lemming must move to next and remain in the walking state.
6. Otherwise (there is no solid floor brick directly below next), the lemming must "step off the ledge" as follows:
   a. The lemming transitions to the falling state.
   b. The lemming resets its fall distance to 0.
   c. The lemming moves horizontally into next immediately (even though there is no floor under next).
7. The lemming must do nothing else during this tick.

Falling State

If the lemming is in the falling state and is not currently idle, then it must do the following:

1. The lemming must ask other actors on the current square if they are a climbable actor (i.e., a net). If so, then the lemming must immediately transition to the climbing state and do nothing more during this tick.
2. Otherwise, the lemming considers the square directly below its current square. Let's call this square "below".
3. If below is not a valid square to move into (either because it is outside the 20×20 world or because it contains a solid floor brick), then the lemming has landed and must resolve the fall:
   a. If the lemming's fall distance is greater than 5 squares, then the lemming dies immediately (see "What a Lemming Must Do When It Dies" below).
   b. Otherwise (fall distance is 5 or less), the lemming survives the fall, transitions back to the walking state and does not move during this tick.
4. Otherwise (below is a valid open square to move into), the lemming must continue falling:
   a. Increment its fall distance by 1.
   b. Move down into below.
5. The lemming must do nothing else during this tick.

Climbing State

If the lemming is in the climbing state and is not currently idle, then during then it must do the following:

1. The lemming must ask other actors on the current square if they are a climbable actor (i.e., a net). If the lemming's current square does not contain a climbable actor (i.e., there is no net on its square), then the lemming must stop climbing:
   a. Transition to the walking state.
   b. Do nothing more during this tick.
2. Otherwise (it is currently on the same square as a net), the lemming attempts to move one square upward. Let's call that square "above".
3. If above is outside the world bounds or contains an object that blocks movement, then the lemming cannot climb higher. It must remain in the climbing state and do nothing during this tick. (It basically gets stuck for the remainder of the level.)
4. Otherwise, the lemming moves up into above and remains in the climbing state.
5. The lemming must do nothing else during this tick.

Bouncing State

If the lemming is in the bouncing state and is not currently idle, it must do the following:

1. The lemming must ask other actors on the current square if they are a climbable actor (i.e., a net). If the lemming's current square contains a climbable actor (i.e., a net), then

the lemming must immediately transition to the climbing state, and do nothing more during this tick.
2. Otherwise, bouncing proceeds as an upward phase followed by an apex phase.
3. While the lemming still has upward steps remaining for this bounce, it attempts to move up by one square during the current tick:
   a. If the square above is inside the world and does not contain an object that blocks movement, the lemming moves up by one square.
   b. Otherwise (out of bounds or an object that blocks movement), the upward phase ends immediately (the lemming does not move upward this tick).
4. As soon as the upward phase is complete (either because it naturally finished the required number of upward steps, or because it ended early due to a blocked upward move), then during the same tick, the lemming performs its apex movement:
   a. It attempts to move one square horizontally in its current facing direction.
   b. If that horizontal square is out of bounds or contains an object that blocks movement, then the lemming does not move horizontally and instead reverses direction (left becomes right, right becomes left) without moving.
   c. Whether or not it successfully moved horizontally, the bounce ends immediately after this apex attempt: the lemming transitions to the falling state and resets its fall distance to 0.

How a Lemming Starts Bouncing

A lemming enters the bouncing state when a spring or trampoline activates on its square. A lemming can be launched only if it is not already bouncing upward; if it is already bouncing, springs and trampolines do not start a new bounce.

When a bounce is initiated, the lemming must track its bounce progress by initializing:
a. A target bounce distance (an integer number of upward steps to travel upward before beginning to fall). This is specified by the trampoline/spring.
b. A counter of how many upward steps it has already attempted during this bounce. This starts out at zero

What a Lemming Must Do When It Dies

A lemming may die when it falls too far (fall distance greater than 5 when it lands), or when it is killed by a hazard such as a bonfire or an ice monster.

When a lemming dies:

1. The lemming must play the SOUND_LEMMING_DIE sound effect.
2. The lemming must inform StudentWorld that one more lemming has died for the level attempt (so the level can fail if too many lemmings die).

3. The lemming must set itself to dead so it will be removed from the world at the end of the tick.

What a Lemming Must Do When It Is Saved

A lemming is saved when it reaches an exit square.

When a lemming is saved:

1. The lemming must set itself to inactive so it will be removed from the world at the end of the tick.
2. The lemming must play the SOUND_LEMMING_SAVED sound effect.
3. A saved lemming must not count as a “dead lemming” for the level failure condition. (In other words, saving a lemming must not increment the dead-lemming counter.)

Separately, when the exit saves a lemming, StudentWorld is responsible for awarding SCORE_SAVED_LEMMING points and incrementing the saved-lemming count used to determine whether the level is complete.

Bonfire

You must create a class to represent a bonfire. Bonfires are stationary hazards that kill lemmings that occupy the same square. Here are the requirements you must meet when implementing the Bonfire class.

What a Bonfire Must Do When It Is Created

When it is first created:

1. A bonfire object must have an image ID of IID_BONFIRE.
2. A bonfire starts at the location specified by the current level’s data file.
3. A bonfire starts facing right.

What a Bonfire Must Do During a Tick

Each time a bonfire is asked to do something (during a tick):

1. If one or more burnable objects (I.e., lemmings) occupy the same square as the bonfire, then the bonfire must kill those lemmings. Each killed lemming must die in the normal way (play SOUND_LEMMING_DIE and count as a dead lemming).
2. The bonfire does not disappear when it kills a lemming. It remains active for the entire level attempt.

A bonfire is not solid; it does not block movement into its square. A bonfire is not launchable from a trampoline or spring, nor is it climbable.

Ice Monster

You must create a class to represent an ice monster. Ice monsters are roaming hazards that patrol left and right on top of floor bricks and kill lemmings that occupy the same square. Here are the requirements you must meet when implementing the IceMonster class.

What an Ice Monster Must Do When It Is Created

When it is first created:

1. An ice monster object must have an image ID of IID_ICE_MONSTER.
2. An ice monster starts at the location specified by the current level's data file.
3. An ice monster starts facing right.

Ice Monster Action Timing

An ice monster does not move every tick. It moves once every 10 ticks. On the ticks where it does not move, it is idle (but can still freeze lemmings!).

What an Ice Monster Must Do During a Tick

Each time an ice monster is asked to do something (during a tick):

1. If a freezable object (I.e., a lemming) occupies the same square as the ice monster, then the ice monster kills the lemming (the lemming dies normally: SOUND_LEMMING_DIE plays and the dead lemming count increases). After killing a lemming in this way, the ice monster does not move during that tick.
2. Otherwise, if the ice monster is idle this tick (because it is not one of its "move ticks"), then it does nothing further.
3. Otherwise, the ice monster attempts to move one square horizontally in the direction it is facing, but only if that movement is "supported" by solid ground:
   a. Let "next" be the square one step left/right in the ice monster's current direction.
   b. If next contains a solid obstruction (a floor brick), then the ice monster does not move, and it reverses direction.
   c. Otherwise, let "belowNext" be the square one step down from next. If belowNext contains a floor brick (solid ground), then the ice monster moves into next.
   d. Otherwise (there is no floor brick below next), the ice monster does not move and instead reverses direction.

An ice monster is not solid; it does not block movement into its square. Lemmings can step onto an ice monster square, and if they do, they will be killed as described above. An ice monster is not launchable from a trampoline or spring, nor is it climbable.

Trampoline

You must create a class to represent a trampoline. Trampolines are tools that can be placed by the player (and may also appear pre-placed in level files). If a lemming that is not already bouncing moves onto the same square as a trampoline, the trampoline bounces the lemming upward in a launch/bounce. Here are the requirements you must meet when implementing the Trampoline class.

What a Trampoline Must Do When It Is Created

When it is first created:

1. A trampoline object must have an image ID of IID_TRAMPOLINE.
2. A trampoline starts at the location specified by the current level's data file, or at the cursor location if placed by the user.
3. A trampoline starts facing right.

What a Trampoline Must Do During a Tick

Each time a trampoline is asked to do something (during a tick):

1. If a bounceable object (I.e., a lemming) occupies the same square as the trampoline AND that lemming is not already in the bouncing state, the trampoline causes that lemming to transition to a bouncing state.
2. The height of the bounce is based on how far the lemming has fallen so far: the lemming's bounce height is set to (its current fall distance minus 1), with the caveat that bounce height is never negative. If the lemming wasn't falling when it goes onto a trampoline, it bounces to a height of 0.
3. When a trampoline successfully launches a lemming, the game must play the SOUND_BOUNCE sound effect.
4. The trampoline does not disappear after bouncing a lemming. It remains in the world and can bounce additional lemmings later.

A trampoline is not solid; it does not block movement into its square. A trampoline is not launchable from a trampoline or spring, nor is it climbable.

Net

You must create a class to represent a net. Nets are tools that can be placed by the player (and may also appear pre-placed in the level files). Nets allow lemmings to climb upward. Here are the requirements you must meet when implementing the Net class.

What a Net Must Do When It Is Created

When it is first created:

1. A net object must have an image ID of IID_NET.
2. A net starts at the location specified by the current level's data file, or at the cursor location if placed by the user.
3. A net starts facing right.

What a Net Must Do During a Tick

Each time a net is asked to do something (during a tick):

1. Nothing… It's just a net

What a Net Must Do Regarding Climbing

A net must be treated as climbable by lemmings. Nets are not solid; they do not block movement into their square. Nets are not launchable from a trampoline or spring.

Important note about climbing behavior: A lemming climbs only while it is on the same square as a net. This means that to build a "ladder," the player generally must place a vertical column of nets (one in each square up the column). If the lemming climbs up into a square that does not contain a net, it will transition to a walking state in the next tick.

One-Way Door

You must create a class to represent a one-way door. One-way doors come in two orientations: left-facing and right-facing. They do not block movement, but they force actors occupying the same square to face the door's direction. Here are the requirements you must meet when implementing the OneWayDoor class.

What a One-Way Door Must Do When It Is Created

When it is first created:

1. A one-way door object must have an image ID of IID_ONE_WAY_DOOR.
2. A one-way door starts at the location specified by the current level's data file, or at the cursor location if placed by the user.

3. A one-way door's direction must be set based on its orientation:
   a. A left-facing door uses direction GraphObject::left.
   b. A right-facing door uses direction GraphObject::right.

What a One-Way Door Must Do During a Tick

Each time a one-way door is asked to do something (during a tick):

1. The one-way door checks whether any actors occupy the same square as the door.
2. If one or more actors occupy the same square as the door, then the one-way door forces those actors to face the door's direction (left or right) by setting their direction accordingly.
3. The one-way door does not disappear after redirecting actors. It remains active for the entire level attempt.

A one-way door is not solid; it does not block movement into its square. A one-way door is not launchable from a trampoline or spring, nor is it climbable.

Pheromone

You must create a class to represent a pheromone. Pheromones are tools that can be placed by the player (and may also appear pre-placed). Pheromones passively attract lemmings horizontally. Here are the requirements you must meet when implementing the Pheromone class.

What a Pheromone Must Do When It Is Created

When it is first created:

1. A pheromone object must have an image ID of IID_PHEROMONE.
2. A pheromone starts at the location specified by the current level's data file, or at the cursor location if placed by the user.
3. A pheromone starts facing right.

What a Pheromone Must Do During a Tick

Each time a pheromone is asked to do something (during a tick):

The pheromone does nothing during its tick. Pheromones are passive objects.

What a Pheromone Must Do Regarding Attraction

A pheromone must be recognized by lemmings as an attractor. During each lemming movement update, pheromones influence lemming direction as described in the lemming section above. Specifically, a pheromone must indicate that it is a lemming attractor if asked by another object

like a lemming. Pheromones are not solid and do not block movement into their square. A pheromone is not launchable from a trampoline or spring, nor is it climbable.

Spring

You must create a class to represent a spring. Springs are tools that can be placed by the player (and may also appear pre-placed). If a lemming that is not already bouncing moves onto the same square as a spring, the spring launches the lemming upward in a launch/bounce. Here are the requirements you must meet when implementing the Spring class.

What a Spring Must Do When It Is Created

When it is first created:

1. A spring object must have an image ID of IID_SPRING.
2. A spring starts at the location specified by the current level's data file, or at the cursor location if placed by the user.
3. A spring starts facing right.

In addition to any other initialization that you decide to do in your Spring class, it must be visible.

What a Spring Must Do During a Tick

Each time a spring is asked to do something (during a tick):

1. If a bounceable object (I.e., a lemming) occupies the same square as the spring AND that lemming is not already in the bouncing state, the spring causes that lemming to transition to a bouncing state.
2. The bounce height when launched by a spring is always 15.
3. When a spring successfully launches a lemming, the game must play the SOUND_BOUNCE sound effect.
4. The spring does not disappear after launching a lemming. It remains in the world and can launch additional lemmings later.

A spring is not solid; it does not block movement into its square. A spring is not launchable from a trampoline or spring, nor is it climbable.

Exit

You must create a class to represent an exit, a destination square where lemmings are saved. Here are the requirements you must meet when implementing the Exit class.

What an Exit Must Do When It Is Created

When it is first created:

1. An exit object must have an image ID of IID_EXIT.
2. An exit starts at the location specified by the current level's data file.
3. An exit starts facing right.

What an Exit Must Do During a Tick

Each time an exit object is asked to do something (during a tick):

1. If one or more lemmings occupy the same square as the exit, then that exit tells one lemming that it has been saved.
2. Saving a lemming means:
   a. The lemming is set to saved and will be removed from the world.
   b. SOUND_LEMMING_SAVED is played.
   c. The player's score increases by 100 points.
   d. The world's saved lemming count increases by 1.

If multiple lemmings occupy an exit square at the same time, that exit saves only one lemming per tick. Remaining lemmings can be saved on subsequent ticks if they remain on the exit. An exit is not solid; it does not block movement into its square. An exit is not launchable from a trampoline or spring, nor is it climbable.

Overall Level Rules: Lemmings, Time, Score, Lives, Winning, and Losing

Each level attempt is a timed simulation puzzle. A level contains exactly 10 lemmings that will be spawned by the factory over time. The player's goal is to save at least half of the level's lemmings by guiding them to the exit using a limited inventory of placeable tools.

Lemming Count and Required Saves

1. Every level spawns exactly 10 lemmings.
2. To complete a level, the player must save at least 5 of them.

Time Limit

1. Each level attempt begins with a timer of 2000 ticks.
2. Each call to StudentWorld::move() corresponds to one tick, and the timer decreases by 1 each tick.

3. If the timer reaches 0 before the player has saved enough lemmings, the player loses one life and must replay the level from scratch (if any lives remain).
4. If the timer reaches 0 and the player has already saved enough lemmings, the level is considered complete immediately.

Deaths and the “Too Many Died” Failure Condition

1. Lemming deaths count toward failing the level attempt. In this project, any lemming that dies (bonfire, ice monster, unsafe fall, etc.) increases the dead lemming count.
2. If strictly more than half of the lemmings die (with 10 lemmings, this means 6 or more deaths), the level attempt immediately fails: the player loses one life and must replay the level from scratch (if any lives remain).

Giving Up

At any time during a level attempt, if the user presses the 'G' key (or 'g'), the player gives up on the current level attempt. Giving up immediately costs one life and restarts the level from scratch (if any lives remain). This give-up behavior is handled by our framework, so there's nothing you need to do.

Lives and Game Over

1. The player begins the game with 3 lives.
2. When the player loses a life (time out, giving up, or too many lemming deaths), the level is restarted from scratch if the player still has at least one remaining life.
3. When the player’s lives reach 0, the game ends.

Advancing to the Next Level and Ending the Game

1. When the player completes a level (by saving enough lemmings under the conditions described above), the game advances to the next level file (level00.txt, then level01.txt, etc.).
2. If the game cannot find the next level file, the game treats this as “no more levels,” and the player has won the game.

Scoring

1. Every lemming that is saved increases the player’s score by 100 points.
2. When the player completes a level, the player also receives a time bonus equal to the number of ticks remaining on the level timer at the moment the level completes. Finishing faster yields a higher score.
3. When the player completes a level, the game must play SOUND_FINISHED_LEVEL.

Tick Processing Order (High-Level)

During each tick of the simulation:

1. The builder cursor is given a chance to act (move and/or place a tool).
2. Every other actor in the world is given a chance to act once (lemmings move, factory spawns, hazards apply effects, etc.).
3. Actors that have died during the tick are removed from the world.

This order is important because the game is a real-time simulation: the world is continuously updating while the player is placing tools.

Object Oriented Programming Best Practices

Before designing your base and derived classes for Project 3 (or for that matter, any other school or work project), make sure to consider the following best practices. These tips will help you not only write a better object-oriented program, but also help you get a better grade on P3!

Try your best to leverage the following best practices in your program, but don't be overly obsessive – it's rarely possible to make a set of perfect classes. That's often a waste of time. Remember, the best is the enemy of the good (enough).

Here we go!

1. You MUST NOT use the imageID (e.g., IID_LEMMING, IID_PLAYER, IID_FLOOR, etc.) to determine the type of an object or store the imageID inside any of your objects as a member variable. You may also not use any other similar approach (e.g., a member string with the object's name, an enumerated type, etc.). YOU MAY NOT USE ANY OTHER CLEVER WAY TO IDENTIFY SPECIFIC TYPES OF OBJECTS THAT WE FORGOT TO MENTION ABOVE :). Doing so will result in a score of ZERO for this project.

2. Avoid using dynamic cast to identify common types of objects. Instead add methods to check for various classes of behaviors:

Don't do this:

```
void decideWhetherToAddOil(Actor* p)
{
  if (dynamic_cast<BadRobot*>(p) != nullptr ||
      dynamic_cast<GoodRobot*>(p) != nullptr ||
      dynamic_cast<ReallyBadRobot*>(p) != nullptr ||
```

```
      dynamic_cast<StinkyRobot*>(p) != nullptr)
      p->addOil();
}
```

Do this instead:

```
void decideWhetherToAddOil (Actor* p)
{
      // define a common method, have all Robots return true, all
      // biological organisms return false
      if (p->requiresOilToOperate())
          p->addOil();
}
```

3. **Always avoid defining specific isParticularClass() methods for each type of object. Instead add methods to check for various common behaviors that span multiple classes:**

Don't do this:

```
void decideWhetherToAddOil (Actor* p)
{
       if (p->isGoodRobot() || p->isBadRobot() || p->isStinkyRobot())
           p->addOil();
}
```

Do this instead:

```
void decideWhetherToAddOil (Actor* p)
{
    // define a common method, have all Robots return true, all
    // biological organisms return false
    if (p->requiresOilToOperate())
      p->addOil();
}
```

4. **If two related subclasses (e.g., SmellyRobot and GoofyRobot) each directly define a member variable that serves the same purpose in both classes (e.g., m_amountOfOil), then move that member variable to the common base class and add accessor and mutator methods for it to the base class. So the Robot base class should have the m_amountOfOil member variable defined once, with getOil() and**

addOil()functions, rather than defining this variable directly in both SmellyRobot and GoofyRobot.

Don't do this:

```
class SmellyRobot: public Robot
       {
                              ...
         private:
           int m_oilLeft;
       };

       class GoofyRobot: public Robot
       {
                              ...
         private:
           int m_oilLeft;
       };
```

Do this instead:

```
class Robot
       {
         public:
           void addOil(int oil) { m_oilLeft += oil; }
           int getOil() const { return m_oilLeft; }
         private:
           int m_oilLeft;
       };
```

5. **Never make any class's data members public or protected. You may make class constants public, protected or private.**

6. **Never make a method public if it is only used directly by other methods within the same class that holds it. Make it private or protected instead.**

7. **Your StudentWorld methods should never return a vector, list or iterator to StudentWorld's private game objects or pointers to those objects. Only StudentWorld should know about all of its game objects and where they are. Instead StudentWorld should do all of the processing itself if an action needs to be taken on one or more game objects that it tracks.**

Don't do this:

```
class StudentWorld
{
  public:
    vector<Actor*> getActorsThatCanBeZapped(Coord coord)
    {
       ...                  // create a vector with actor pointers and return it
    }
};

class NastyRobot
{
  public:
    virtual void doSomething()
    {
      vector<Actor*> v;
      vector<Actor*>::iterator p;

      v = studentWorldPtr->getActorsThatCanBeZapped(getCoord());
      for (p = actors.begin(); p != actors.end(); p++)
        p->zap();

    }
};
```

Do this instead:

```
class StudentWorld
{
  public:
    void zapAllZappableActors(Coord coord)
    {
      for (p = actors.begin(); p != actors.end(); p++)
        if (p->getCoord() == coord && p->isZappable())
          p->zap();
    }
};

class NastyRobot
{
  public:
    virtual void doSomething()
    {
      studentWorldPtr->zapAllZappableActors(getCoord());
    }
};
```

8. If two subclasses have a method that shares some common functionality, but also has some differing functionality, use an auxiliary method to factor out the differences:

Don't do this:

```
class StinkyRobot: public Robot
{
                ...
  public:
    virtual void doDifferentiatedStuff()
    {
      doCommonThingA();
      passStinkyGas();
      pickNose();
      doCommonThingB();
    }
};

class ShinyRobot: public Robot
{
                ...
  public:
    virtual void doDifferentiatedStuff()
    {
      doCommonThingA();
      polishMyChrome();
      wipeMyDisplayPanel();
      doCommonThingB();
    }
};
```

Do this instead:

```
class Robot
{
  public:
    virtual void doSomething()
    {
        // first do the common thing that all robots do
      doCommonThingA();

        // then call a virtual function to do the differentiated stuff
      doDifferentiatedStuff();

        // then do the common final thing that all robots do
      doCommonThingB();
    }

  private:
```

```
    virtual void doDifferentiatedStuff() = 0;
};

class StinkyRobot: public Robot
{
                ...
  private:
                // define StinkyRobot's version of the differentiated function
    virtual void doDifferentiatedStuff()
    {
        // only Stinky robots do these things
      passStinkyGas();
      pickNose();
    }
};

class ShinyRobot: public Robot
{
                ...
  private:
      // define ShinyRobot's version of the differentiated function
    virtual void doDifferentiatedStuff()
    {
        // only Shiny robots do these things
      polishMyChrome();
      wipeMyDisplayPanel();
    }
};
```

Yes, it is legal for a derived class to override a virtual function that was declared private in the base class. (It's not trying to *use* the private member function; it's just defining a new function.)

Don’t know how or where to start? Read this!

When working on your first large object-oriented program, you’re likely to feel overwhelmed and have no idea where to start; in fact, it’s likely that many students won’t be able to finish their entire program. Therefore, it’s important to attack your program piece by piece rather than trying to program everything at once.

Students who try to program everything at once rather than program incrementally almost always fail to solve CS32’s project 3, so don’t do it!

Instead, try to get one thing working at a time. Here are some hints:

1. When you define a new class, try to figure out what public member functions it should have. Then write dummy “stub” code for each of the functions that you’ll fix later:

```
class Foo
{
  public:
       int chooseACourseOfAction() { return 0; }   // dummy version
};
```

Try to get your project compiling with these dummy functions first, then you can worry about filling in the real code later.

2. Once you’ve got your program compiling with dummy functions, then start by replacing one dummy function at a time. Update the function, rebuild your program, test your new function, and once you’ve got it working, proceed to the next function.
3. **Make backups of your working code frequently. Any time you get a new feature working, make a backup of all your .cpp and .h files just in case you screw something up later.**

BACK UP YOUR .CPP AND .H FILES TO A REMOVABLE DEVICE OR TO ONLINE STORAGE JUST IN CASE YOUR COMPUTER CRASHES!

If you use this approach, you’ll always have something working that you can test and improve upon. If you write everything at once, you’ll end up with hundreds of errors and just get frustrated! So don’t do it.

Building the Game

The game assets (i.e., image, sound, and level data files) are in a folder named *Assets*. The way we’ve written the main routine, your program will look for this folder in a standard place (described below for Windows and Mac OS X). A few students may find that their environment is set up in a way that prevents the program from finding the folder. If that happens to you, change the string literal `"Assets"` in main.cpp to the full path name of wherever you choose to put the folder (e.g., `"C:/proj3/Assets"` or `"/Users/fred/proj3/Assets"`).

To build the game, follow these steps:

For Windows

Unzip the Lemmings-skeleton-windows.zip archive into a folder on your hard drive. Double-click on Lemmings.sln to start Visual Studio.

If you build and run your program from within Visual Studio, the Assets folder should be in the same folder as your *.cpp* and *.h* files. On the other hand, if you launch the program by double-clicking on the executable file, the Assets folder should be in the same folder as the executable.

For Mac OS X

Unzip the Lemmings-skeleton-mac.zip archive into a folder on your hard drive.
Double-click on our provided Lemmings.xcodeproj to start Xcode.

If you build and run your program from within Xcode, the Assets directory should be in the directory *yourProjectDir*/DerivedData/*yourProjectName*/Build/Products/Debug (e.g., /Users/fred/Lemmings/DerivedData/Lemmings/Build/Products/Debug). On the other hand, if you launch the program by double-clicking on the executable file, the Assets directory should be in your home directory (e.g., */Users/fred*).

What to Turn In

Part #1 (20%)

Ok, so we know you're probably feeling overwhelmed and aren't sure where to begin. That's completely normal for a project of this size. To help with that, we're deliberately structuring Project 3 so you work incrementally instead of trying to build the entire game at once. For Part 1, your job is to build a very small, self-contained slice of the Lemmings game that implements only a fraction of the final behavior. Think of this as laying down the core scaffolding that everything else will eventually build on.

For Part 1, you must program the following pieces.

First, you must define a class that serves as the base class for all actors that exist in the world, such as floor bricks, ice monsters, lemmings, bonfires, tools, and so on. This base class must be derived from GraphObject. It must have a simple constructor, and it must define a virtual member function named doSomething() that can be called once per game tick to let the actor take its action. You are free to add additional public or private member functions and data members as needed, as long as you follow good object-oriented design and avoid duplicating functionality across subclasses.

Second, you must implement a FloorBrick class, derived (directly or indirectly) from your base Actor class. The FloorBrick must have a simple constructor and must use the image ID IID_FLOOR. Floor bricks do not do anything during a tick, so their doSomething() method can

be empty. You may add any additional helper methods or data members that make sense, provided you keep your design clean.

Third, you must implement an IceMonster class, also derived from your Actor base class. This is a very limited version of the ice monster used in the full game. The ice monster must use the appropriate image ID for an ice monster (IID_ICE_MONSTER), and must move left and right along the platform it starts on. On each tick, its doSomething() method should attempt to move one square in its current horizontal direction. If moving forward would cause it to walk into a wall or step off the edge of its platform, it should reverse direction and remain in its current square for that tick. For Part 1, the ice monster does nothing else: it does not interact with lemmings, does not cause damage, and does not respond to any other game elements.

Fourth, you must implement a limited version of the StudentWorld class. This class is responsible for managing the world and the actors in it. You should add whatever private data members are necessary to keep track of all floor bricks and ice monsters in the level. You may ignore all other game elements such as lemmings, exits, bonfires, tools, scoring, lives, or time limits for Part 1.

Your StudentWorld constructor should initialize your data members, and your destructor must free any dynamically allocated memory that still exists when the StudentWorld object is destroyed. You must implement the init() method so that it reads the current level file and creates FloorBrick and IceMonster objects at the appropriate grid locations specified by the level data. You must also implement the move() method so that, during each tick, it asks every actor in the world to doSomething(). For Part 1, you do not need to check win or loss conditions, and you do not need to worry about lemmings being created or dying. Finally, you must implement a cleanUp() method that deletes all dynamically allocated actors created during init() or move(). Even if your destructor simply calls cleanUp(), both functions must exist.

As you work on these classes, build your program frequently. You will almost certainly see a lot of compiler errors at first. That’s expected. Fix them incrementally and keep going. Large projects always look intimidating until the first few pieces start working.

You’ll know you’re done with Part 1 when your program builds successfully and does the following: when the game starts, the level displays all floor bricks in their correct locations, along with any ice monsters specified in the level file. As the game runs, each ice monster should visibly patrol back and forth along its platform, reversing direction when it reaches a wall or the edge.

Your Part 1 solution is allowed to do more than what is specified above; for example, if you are further along in the project, and what you have builds and has at least as much functionality as what’s described above, then you may turn that in instead.

For Part 1, you are not required to implement lemmings, lemming factories, exits, bonfires, tools, scoring, or player interaction of any kind. You may add any of these features if you wish, but they are not required. If you do add extra functionality, make sure that your FloorBrick,

IceMonster, Actor, and StudentWorld classes still behave correctly and that your program still builds and satisfies the Part 1 requirements.

If you can get this small version working, you will have completed most of the difficult architectural thinking for the project. The full game will require additional logic, but it will mostly build on top of the structure you create here.

What to turn in for Part 1: you must submit your source code for this simple version of the game, and it must build without errors under either Visual Studio or Xcode. You do not need to support multiple compilers. You will submit a zip file containing exactly these four files:

Actor.h, which contains the declarations for your Actor base class, FloorBrick class, and IceMonster class, along with any required constants.

Actor.cpp, which contains the implementations of those classes.

StudentWorld.h, which contains your StudentWorld class declaration.

StudentWorld.cpp, which contains your StudentWorld class implementation.

Do not submit any other files. We will test your code using our versions of all other provided files. You must not modify any of our files, or you will receive zero credit, with the sole exception that you may modify the string literal "Assets" in main.cpp if necessary. There is no written report for Part 1. We will not evaluate comments or documentation at this stage. For Part 1, only correct behavior for the specified subset of functionality matters.

What to Turn In For Part #2

You must turn in your source code for your game, which **must build without errors** (warnings are OK) under either Visual Studio or Xcode. You do **not** have to get it to run under more than one compiler. You will turn in a zip file containing nothing more than these five files:

```
Actor.h              // contains declarations of your actor classes
                     //     as well as constants required by these classes
Actor.cpp            // contains the implementation of these classes
StudentWorld.h       // contains your StudentWorld class declaration
StudentWorld.cpp     // contains your StudentWorld class implementation

report.docx or report.txt  // your report (5% of your grade)
```

You will not be turning in any other files – we'll test your code with our versions of the other .cpp and .h files. Therefore, your solution must NOT modify any of our files or you will receive zero credit! (Exception: You may modify the string literal "Assets" in main.cpp.)

You must turn in a report that contains the following:

1. A description of the control flow for the interaction of a lemming and a trampoline. Where in the code is the co-location of the two objects detected, and what happens from that point until the interaction is finished? Which functions of which objects are called and what do they do during the handling of this situation?
2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. “I didn’t implement the exit class.” or “My Trampoline doesn’t work correctly yet so I treat it like a Spring right now.”
3. A list of other design decisions and assumptions you made; e.g., “It was not specified what to do in situation X, so this is what I decided to do.”

FAQ

Q: The specification is silent about what to do in a certain situation. What should I do?
A: Play with our sample program and do what it does. Use our program as a reference. If neither the specification nor our program makes it clear what to do, do whatever seems reasonable and document it in your report. **If the specification is unclear, but your program behaves like our demonstration program, YOU WILL NOT LOSE POINTS!**

Q: What should I do if I can’t finish the project?!
A: Do as much as you can, and whatever you do, make sure your code builds! If we can sort of play your game, but it’s not complete or perfect, that’s better than its not even building!

Q: Where can I go for help?
A: Try TBP/HKN/UPE – they provide free tutoring and can help you with your project!

Q: Can I work with my classmates on this?
A: You can discuss general ideas about the project, but don’t share source code with your classmates. Also don’t help them write their source code.

GOOD LUCK!