Project 3
Marble Madness

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

Time due:

Table of Contents

Introduction

NachenGames corporate spies have learned that SmallSoft is planning to release a new game called Marble Madness, and would like you to program an exact copy so NachenGames can beat SmallSoft to the market. To help you, NachenGames corporate spies have managed to steal a prototype Marble Madness executable file and several source files from the SmallSoft headquarters, so you can see exactly how your version of the game must work (see posted executable file) and even get a head start on the programming. Of course, such behavior would never be appropriate in real life, but for this project, you'll be a programming villain.

In Marble Madness, the player has to navigate through a series of robot-infested mazes in order to gather valuable crystals. After the player has gathered each of the crystals within a particular maze, an exit will be revealed, and the player may then use the exit to advance to the next maze. The player wins by completing all of the mazes.

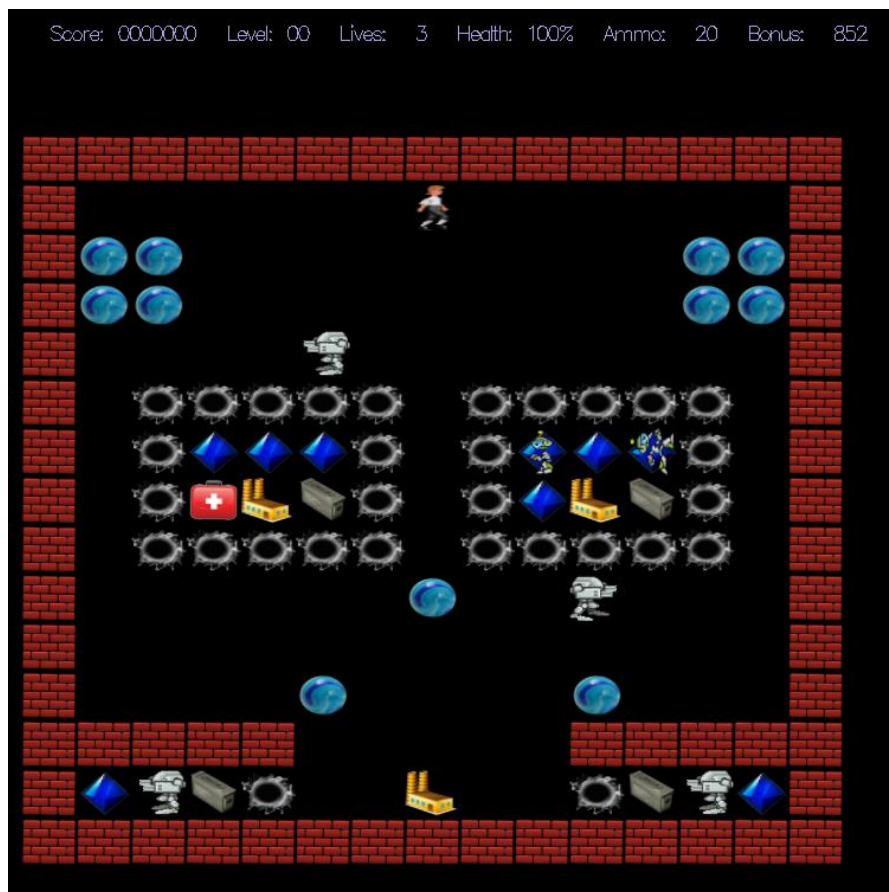Here is an example of what the Marble Madness game looks like:

Figure #1: A screenshot of the Marble Madness game. You can see the player (top-middle), two different types of robots (RageBots and ThiefBots), a bunch of marbles, a bunch of pits (that can only be crossed if they are filled in with a marble), ThiefBot factories, blue crystals, a healing kit, and an ammo kit.

Game Details

In Marble Madness, the player starts out a new game with three lives and continues to play until all of his/her lives have been exhausted. There are multiple levels in Marble Madness, beginning with level 0, and each level has its own maze. During each level, the player must gather all of the blue crystals within the current maze before the exit is revealed and they may use it to move on to the next level.

Upon starting each level, the player's avatar is placed in a maze filled with one or more blue crystals, marbles, pits, robots, robot factories and other goodies. The player may use the arrow keys to move their avatar (the Indiana Jones-looking character) left, right, up and down through the maze. They may walk on any square so long as it doesn't have a wall, a marble, a factory, a robot, or a pit on it. The player may walk onto a square containing a marble if they are able to push it out of the way first. In addition to walking around the maze, the player may also shoot their pea cannon – but beware, it has only limited peas. Peas can destroy robots as well as marbles, but it takes more than one shot to do so.

There are three different types of goodies distributed throughout the maze that the player can collect in addition to blue crystals. If the player's avatar steps upon the same square as an extra life goodie, it instantly gives the player an extra life. If the avatar steps onto the same square as a restore health goodie, it will restore the player to full health (in case of having been injured by shots from the robots). Finally, if the avatar steps onto the same square as an ammo goodie, it will give the player 20 additional peas.

There are four major types of robots in Marble Madness: Horizontal RageBots, Vertical RageBots, Regular ThiefBots and Mean ThiefBots

As mentioned, RageBots fall into two categories – Horizontal RageBots and Vertical RageBots. Horizonal RageBots simply move back and forth on a row of the screen (only reversing course when they run into an obstacle), shooting at the player's avatar if he/she ever walks in their line of sight. Vertical RageBots are identical to their Horizontal cousins, but move up and down within a single column of the maze. RageBots can start out anywhere in the maze (depending on where the designer of the maze choses to put them).

In contrast to the RageBot, the ThiefBots are a bit nastier. These robots wander around the maze looking for goodies to steal. If they happen to step onto a goodie (an extra life goodie, a restore health goodie, or an ammo goodie) and they don't already hold one, they may pick it up for themselves. As with RageBots, there are two types of ThiefBots: Regular ThiefBots and Mean ThiefBots. Regular ThiefBots simply wander aimlessly around the maze looking for, and picking up, goodies. They are otherwise harmless and will not fire upon the player's avatar. In contrast, in addition to picking up goodies, Mean ThiefBots will fire a pea anytime the player steps in their path. So beware! When any type of ThiefBot dies, if it previously picked up a goodie, it will drop this object upon its square in the maze.

ThiefBots are created by ThiefBot factories, of which there are two types – one that produces Regular ThiefBots and one that produces Mean ThiefBots. ThiefBots never start out in the maze; they are added only by factories.

Once the player has collected all of the blue crystals within the current maze, an exit will appear. The exit is invisible and unusable until all of the crystals have been collected from the level. Once the exit has been revealed, the player must direct their avatar to the exit in order to advance to the next level. The player will be granted 2000 points for exiting a level. The player will also be given a bonus for completing the level quickly, if they did so fast enough. The game is complete once the player has used the exit on the last level.

If the player's health reaches zero (the player loses health when shot), their avatar dies and loses one "life." If, after losing a life, the player has one or more remaining lives left, they are placed back on the current level and they must again solve the entire level from scratch (with the level starting as it was at the beginning of the first time it was attempted). The player will restart the level with full health points, as well as 20 peas (regardless of how many they had when they died). If the avatar dies and has no lives left, then the game is over.

The Marble Madness maze is exactly 15 squares wide by 15 squares high, and both the player's avatar and robots may move to any adjacent square that doesn't contain a wall, a pit, a marble, a robot, the player or a factory. The one exception is that ThiefBots are born in Factories, so they may start out on the same square as a factory, but are not allowed to move back onto their birth factory once they've left it. The bottom-leftmost square has coordinates x=0,y=0, while the upper-rightmost square has coordinate x=14,y=14, where x increases to the right and y increases upward toward the top of the screen. You can look in our provided file, *GameConstants.h*, for constants that represent the maze's width and height (VIEW_WIDTH and VIEW_HEIGHT).

In Marble Madness, each level's maze is stored in a different data file. For example, the first level's maze is stored in a file called level00.txt. The second level's maze is stored in a file called level01.txt, and so on. Each time the player is about to start a new level, your code must load the data from the appropriate data file (using a class called *Level* that we provide) and then use this data to determine the layout of the current level.

Each level data file contains a specification for the layout of the maze, the initial locations of all the RageBots and the player's avatar, as well as the initial locations of all marbles, pits, factories, crystals, goodies and the exit. For more information on the level data files, please see the Level Data File section below. You may define your own level data files to customize your game (and more importantly, to test it).

Once a new maze has been prepared and the player's avatar and all the robots and items in the maze have been properly situated, the game play begins. Game play is divided into

ticks, and there are twenty ticks per second (to provide smooth animation and game play). During each tick, the following occurs:

1. The player has an opportunity to move their avatar exactly one square horizontally or vertically, fire their pea cannon (if they have peas), or give up (some levels are unsolvable if the player makes a mistake, so if the player realizes this, they can press the Escape key to lose a life and restart the level from scratch).
2. Every other object in the maze (e.g., RageBots, ThiefBots, factories, goodies, etc.) is given an opportunity to do something. For example, when given the opportunity to do something, a RageBot can move one square (left, right, up or down) according to its built-in movement algorithm (the RageBot movement algorithms are described in detail in the various RageBot sections below).

The player controls the direction of their avatar with the arrow keys, or for lefties and others for whom the arrow key placement is awkward, WASD or the numeric keypad: up is *w* or *8*, left is *a* or *4*, down is *s* or *2*, right is *d* or *6*. The player may move their avatar between the maze's Walls as they please. The player can sacrifice one life and restart the current level by pressing the Escape key at any time.

The player's avatar may fire their pea cannon by pressing the space bar. A pea that hits a RageBot or a ThiefBot does 2 points of damage to it. If, after repeated hits, the robot dies, the player earns points:

For destroying a RageBot of any type:	100 points
For destroying a Regular ThiefBot:	10 points
For destroying a Mean ThiefBot:	20 points

The player also earns points (and special benefits) by picking up (i.e., moving onto the same square as) various items:

Blue Crystal: 50 points (all crystals must be collected to advance to the next level)
Extra Life Goodie: 1000 points (the user gets an extra life)
Restore Health Goodie: 500 points (the user's health is restored to 100%)
Ammo Goodie: 100 points (the user receives 20 additional peas)

Players also earn bonus points for completing a level quickly. Each level's maze starts with a bonus score of 1000 points. During each tick of the game, the bonus score is reduced by one point until the bonus reaches zero (the bonus never goes below zero). If and when the player completes the current level, whatever bonus remains is added onto their score. This incentivizes the player to complete each level as quickly as possible since the player wants to maximize their score in the game.

The player starts with three lives. The player loses a life if their health reaches zero (from being shot by robots).

When a Player dies, the player's number of remaining lives is decremented by 1. If the player still has at least one life left, then the user is prompted to continue and given another chance by restarting the current maze level from scratch. All the RageBots that were initially on the level will again be alive and returned to their starting positions, the player's avatar will be returned to their starting position, and the maze will revert back to its original state (all crystals, goodies, pits and marbles in their initial positions). In addition, if the exit was exposed in the maze prior to the avatar's death, then this too will be hidden from the maze until such time that the player collects all Crystals on the level. Then game play restarts. If the player is killed and has no lives left, then the game is over. Pressing the *q* key lets you quit the game prematurely.

So how does a video game work?

Fundamentally, a video game is composed of a bunch of objects; in Marble Madness, those objects include the player's avatar, RageBots (Horizontal and Vertical), ThiefBots (Regular and Mean), goodies (e.g., extra life goodies), crystals, marbles, pits, walls, factories, 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 maze, its own internal state (e.g., a RageBot knows its location, what direction it's moving, 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 player's avatar, the algorithm that controls the avatar actor object is the user's own brain and hand, and the keyboard! In the case of other actors (e.g., RageBots), each object has an internal autonomous algorithm and state that dictates how the object behaves in the game world.

Once a game begins, gameplay is divided into *ticks*. A tick is a unit of time, for example, 50 milliseconds (that's 20 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 1 square to the left), or change other objects' states (e.g., when a RageBot's algorithm is called by the game, it may determine that the player's avatar has moved into its line of fire, and it may fire its pea cannon). 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 player. For example, a RageBot will move just one square left/right/up/down, rather than moving two or more squares; a RageBot moving, say, 5 squares in a single tick would confuse the user, because humans are used to seeing smooth movement in video games, not jerky shifts.

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 (that we provide) animates the actors onto the screen in their new configuration. So, if a RageBot changed its location from 10,5 to 11,5 (moved one square right), then our game framework would erase the graphic of the RageBot from location 10,5 on the screen and draw the RageBot's graphic

at 11,5 instead. Since this process (asking actors to do something, then animating them to the screen) happens 20 times per second, the user will see somewhat 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., a RageBot doesn't move 3 inches away from where it was during the last tick, but instead moves 1 millimeter away), 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 game World is initialized and prepared for play. This involves allocating one or more actors (which are C++ objects) and placing them in the game world so that they will appear in the maze.

Game play: Game play is broken down into a bunch of ticks. During each tick, all of the actors in the game have a chance to do something, and perhaps die. During a tick, new actors may be added to the game and actors who die must be removed from the game 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., robots, walls, marbles, goodies, the player's avatar, etc.) since the level has ended. If the game is not over (i.e., the player has more lives), then the game proceeds back to the *Initialization* step, where the maze is repopulated with new occupants and game play restarts at the current level.

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

```
while (The player has lives left)
{
    Prompt_the_user_to_start_playing(); // "press a key to start"
    Initialize_the_game_world();         // you're going to write this

    while (the player is still alive)
    {
        // each pass through this loop is a tick (1/20th of a sec)

        // 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_for_50ms_to_give_the_user_time_to_react();
    }
```

```
        // the player died - you're going to write this code
        Cleanup_all_game_world_objects();   // you're going to write this
        if (the player is still alive)
            Prompt_the_Player_to_continue();
    }

    Tell_the_user_the_game_is_over();        // we provide this
```

And here is what the `Ask_all_actors_to_do_something()` function might look like:

```
  void Ask_all_actors_to_do_something()
  {
     for each actor 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. For example, here is some pseudocode showing what a (simplified) RageBot might decide to do each time it gets asked to do something:

```
  class RageBot: public SomeOtherClass
  {
     public:
         void doSomething()
         {
             If the player is in my line of sight, then
                     Fire my pea cannon in the direction of the player
             Else if I can move in my current direction w/o hitting an obstacle, then
                     Move one square in my current direction
             Else if I'm about to run into an obstacle, then
                     Reverse my direction, but don't move during this tick
         }
         ...
  };
```

And here's what the player's *doSomething( )* member function might look like:

```
  class Player: public …
  {
     public:
         void doSomething()
         {
              Try to get user input (if any is available)
                If the user pressed the UP key and that square is open then
                    Increase my y location by one
                If the user pressed the DOWN key and that square is open then
                    Decrease my y location by one
                ...
                If the user pressed the space bar to fire and the player has
                  peas, then
                    Introduce a new pea object into the game
                ...
         }
         ...
  };
```

What Do You Have to Do?

You must create a number of different classes to implement the Marble Madness game. Your classes must work properly with our provided classes, and **you must not modify our classes or our source files in any way** to get your classes to work properly (**doing so will result in a score of zero on the entire project!**). Here are the specific classes that you must create:

1. You must create a class called *StudentWorld* which is responsible for keeping track of your game world (including the maze) and all of the actors/objects (RageBots, ThiefBots, peas, crystals, goodies, pits, marbles, the player's avatar, etc.) that are inside the maze.
2. You must create a class to represent the player in the game.
3. You must create classes for Horizontal/Vertical RageBots, Regular ThiefBots, Mean ThiefBots, factories, marbles, pits, walls, crystals, extra life goodies, restore health goodies, ammo Goodies, walls, and the exit, as well as any additional base classes (e.g., a robot base class if you find it convenient) that help you implement the game.

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 maze and all of its inhabitants such as RageBots, ThiefBots, the player, marbles, walls, the exit, goodies, etc.). It is responsible for initializing the game world at the start of the game, asking all the actors to do something during each tick of the game, destroying an actor when it disappears (e.g., a RageBot dies), and destroying all of the actors in the game world when the user loses a life.

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;
```

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.

When a new level starts (e.g., at the start of a 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 maze from a data file (we'll show you how below), 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 (a) when the game first starts, (b) when the player completes the current level and advances to a new level (that needs to be loaded/initialized), or (c) when the user loses a life (but has more lives left) and the game is ready to restart at the current level.

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, or if the level just completed is level 99, 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.

Once a new level has been loaded/initialized with a call to the *init()* method, our game framework will repeatedly call the *StudentWorld's move()* method, at a rate of roughly 20 times per second. 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 game actors (e.g., the player's avatar, each RageBot, goodie, etc.) to try to do something: e.g., move themselves and/or perform their specified behavior. Finally, this method is responsible for disposing of (i.e., deleting) actors (e.g., a pea, a dead RageBot, etc.) that need to disappear during a given tick. For example, if a RageBot is shot by the player and its "hit points" (life force) drains to zero, then its state should be set to dead, and then after all of the actors in the game get a chance to do something during the tick, the *move()* method should remove that RageBot from the game world (by deleting its object and removing any reference to the object from the *StudentWorld's* data structures). The *move()* method will automatically be called once during each tick of the game by our provided game framework. You will never call the *move()* method yourself.

The *cleanup()* method is called by our framework when the player completes the current level or loses a life (i.e., her/his hit points reach zero due to being shot by the robots). The *cleanup()* method is responsible for freeing all actors (e.g., all RageBot objects, all ThiefBot objects, all wall objects, the Avatar object, the exit object, all goodie objects, crystal objects, all pea objects, etc.) that are currently in the game. This includes all actors created during either the *init()* method or introduced during subsequent game play by the actors in the game (e.g., a ThiefBot that was added to the maze by a ThiefBot factory) that have not yet been removed from the game.

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:

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

getLevel() can be used to determine the current level number.

getLives() can be used to determine how many lives the player has left.

decLives() reduces the number of Player lives by one.

incLives() increases the number of Player lives by one.

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

increaseScore() is used by a *StudentWorld* object (or your other classes) to increase the user's score upon successfully destroying a robot, picking up a goodie of some sort, or completing a level (to give the player their remaining level bonus). When your code calls this method, you must specify how many points the user gets (e.g., 100 points for destroying a RageBot). This means that the game score is controlled by our *GameWorld* object – you must *not* maintain your own 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, e.g.:

```
Score: 0321000  Level: 05  Lives:  3  Health:  70%  Ammo:  20  Bonus:  742
```

assetPath() returns the path to 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 to move the player, to fire, or to sacrifice one life and restart the level. This method returns true if the user hit a key during the current tick, and false otherwise (if the user did not hit any key during this tick). The only 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). If the function returns true, the argument will be set to one of the following values (defined in *GameConstants.h*):

```
KEY_PRESS_LEFT
KEY_PRESS_RIGHT
```

```
KEY_PRESS_UP
KEY_PRESS_DOWN
KEY_PRESS_SPACE
KEY_PRESS_ESCAPE
```

The *playSound()* method can be used to play a sound effect when an important event happens during the game (e.g., a robot dies or the player picks up a crystal). You can find constants (e.g., SOUND_ROBOT_DIE) that describe what noise to make in the *GameConstants.h* file. Here's how this method might be used:

```
    // if a RageBot reaches zero hit points and dies, make a dying sound

if (theRobotHasZeroHitPoints())
        studentWorldObject->playSound(SOUND_ROBOT_DIE);
```

init() Details

Your StudentWorld's *init()* member function must:

1. Initialize the data structures used to keep track of your game's world.
2. Load the current maze details from a level data file.
3. Allocate and insert a valid Avatar object into the game world.
4. Allocate and insert any RageBots objects, wall objects, marble objects, factory objects, crystal objects, goodie objects, or exit objects into the game world, as required by the specification in the current level's data file.

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

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

int StudentWorld::someFunctionYouWriteToLoadALevel()
{
        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 – here's an example

        int x = 0;
        int y = 5;
        Level::MazeEntry item = lev.getContentsOf(x, y);
        if (item == Level::player)
                cout << "The player should be placed at 0,5 in the maze\n";
        x = 10;
        y = 7;
```

```
                    item = lev.getContentsOf(x, y);
                    if (item == Level::wall)
                            cout << "There should be a wall at 10,7 in the maze\n":

                    … // etc
            }
```

Notice that the *getContentsOf()* method takes the column parameter (x) first, then the row parameter (y) second. This is different than the order one normally uses when indexing a 2-dimensional array, which would be array[row][col]. Be careful!

You can examine the *Level.h* file for a full list of functions that you can use to access each level.

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 a *StudentWorld* object. It is **required** that you keep track of all of the actors (e.g., RageBots, walls, extra life goodies, the exit, crystals, marbles, etc.) in a **single** STL collection like a *list* or *vector*. (To do so, we recommend using a container of pointers to the actors). If you like, your *StudentWorld* object may keep a separate pointer to the Avatar object rather than keeping a pointer to that object in the container with the other actor pointers; the player is the **only** actor allowed to not be stored in the single actor container.

You must not call the *init()* method yourself. Instead, this method will be called by our framework code when it's time for a new game to start (or when the player completes a level or needs to restart a level).

move() Details

The *move()* method must perform the following activities:

1. It must ask all of the actors that are currently active in the game world to do something (e.g., ask a RageBot to move itself, ask a factory to potentially produce a new ThiefBot, give the player a chance to move up, down, left or right, etc.).
    a. If an actor does something that causes the player to die, then the *move()* method should immediately return GWSTATUS_PLAYER_DIED.
    b. If the player steps onto the same square as an exit (after first collecting all of the crystals on the level), completing the current level, then the *move()* method should immediately:
        i. Increase the player's score appropriately (by 2000 points for using the exit, and by the remaining bonus score for the level).
        ii. Return a value of GWSTATUS_FINISHED_LEVEL.
2. It must then delete any actors that have died during this tick (e.g., a RageBot that was killed by a pea so both should be removed from the game world, or a goodie that disappeared because the player picked it up).
3. It must reduce the level's bonus points by one point during each tick. Each level starts with a bonus of 1000, then goes down during each tick. So, after the first

tick, the bonus would drop to 999, after the second tick to 998, etc. This declining bonus value incentivizes the player to complete the level as quickly as possible to get the biggest bonus score. The level bonus may not go below a value of zero.
4. It should expose/activate the exit on the current level once the player has collected all of the crystals on the level (alternatively, the exit object can do this instead of the *StudentWorld* object). This enables the player to later move over to the exit and complete the level, advancing to the next level.
5. It must update the status text on the top of the screen with the latest information (e.g., the user's current score, the remaining bonus score for the level, etc.).

The *move()* method must return one of three different values when it returns at the end of each tick (all are defined in *GameConstants.h*):

```
GWSTATUS_PLAYER_DIED
GWSTATUS_CONTINUE_GAME
GWSTATUS_FINISHED_LEVEL
```

The first return value indicates that the player died during the current tick, and instructs our provided framework code to tell the user the bad news and restart the level if the user has more lives left. If your *move()* method returns this value and the player has more lives left, then our framework will prompt the player to continue the game, call your *cleanup()* method to destroy the level, call your *init()* method to re-initialize the level from scratch, and then begin calling your *move()* method over and over, once per tick, to let the user play the level again.

The second return value indicates that the tick completed without the player dying BUT the player has not yet completed the current level. Therefore, the game play should continue normally for the time being. In this case, the framework will advance to the next tick and call your *move()* method again.

The final return value indicates that the player has completed the current level (that is, gathered all of the crystals on the level and stepped onto the same square as the exit). If your *move()* method returns this value, then the current level is over, and our framework will call your *cleanup()* method to destroy the level, advance to the next level, then call your *init()* method to prepare that level for play, etc…

Here's pseudocode for how the move() method might be implemented:

```
int StudentWorld::move()
{
    // Update the Game Status Line
    updateDisplayText();   // update the score/lives/level text at screen top

    // The term "actors" refers to all robots, the player, Goodies,
    // Marbles, Crystals, Pits, Peas, the exit, etc.

    // Give each actor a chance to do something
    for each of the actors in the game world
    {
        if (actor[i] is still active/alive)
```

```
            {
                 // ask each actor to do something (e.g. move)
                actor[i]->doSomething();

                if (thePlayerDiedDuringThisTick())
                        return GWSTATUS_PLAYER_DIED;

                if (thePlayerCompletedTheCurrentLevel())
                {
                        increaseScoreAppropriately();
                        return GWSTATUS_FINISHED_LEVEL;
                }
            }
        }

        // Remove newly-dead actors after each tick
        removeDeadGameObjects(); // delete dead game objects

        // Reduce the current bonus for the Level by one
        reduceLevelBonusByOne();

        // If the player has collected all of the crystals on the level, then we
        // must expose the exit so the player can advance to the next level
        if (thePlayerHasCollectedAllOfTheCrystalsOnTheLevel())
                exposeTheExitInTheMaze();  // make the exit Active

        // return the proper result
        if (thePlayerDiedDuringThisTick())
                return GWSTATUS_PLAYER_DIED;

        if (thePlayerCompletedTheCurrentLevel())
        {
                increaseScoreAppropriately();
                return GWSTATUS_FINISHED_LEVEL;
        }

        // the player hasn't completed the current level and hasn't died, so
        // continue playing the current level
        return GWSTATUS_CONTINUE_GAME;
    }
```

Give Each Actor a Chance to Do Something

During each tick of the game each active actor must have an opportunity to do something (e.g., move around, shoot, etc.). Actors include the player's avatar, RageBots, ThiefBots, walls, crystals, marbles, factories, goodies, pits, peas, and the exit.

Your *move()* method must enumerate each active actor in the maze (i.e., held by your *StudentWorld* object) and ask it to do something by calling a member function in the actor's object named *doSomething()*. In each actor's *doSomething()* method, the object will have a chance to perform some activity based on the nature of the actor and its current state: e.g., a RageBot might move one step forward, the player might shoot a pea, a marble may fill a pit in the maze, etc.

It is possible that one actor (e.g., a pea) may destroy another actor (e.g., a RageBot) during the current tick. If an actor has died earlier in the current tick, then the dead actor must not have a chance to do something 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.

Remove Dead Actors after Each Tick

At the end of each tick, your *move()* method must determine which of your actors are no longer alive, remove them from your container of active actors, and delete their objects (so you don't have a memory leak). So if, for example, a RageBot's hit points go to zero (due to it being shot) and it dies, then it should be noted as dead, and at the end of the tick, its *pointer* should be removed from the StudentWorld's container of active objects, and the RageBot object should be deleted (using the C++ delete expression) to free up memory for future actors that will be introduced later in the game. (Hint: Each of your actors could have a data member indicating whether or not it is still alive)

Updating the Display Text

Your *move()* method must update the game statistics at the top of the screen during every tick by calling the *setGameStatText()* method that we provide in our *GameWorld* class. You could do this by calling a function like the one below from the *move()* method:

```
void setDisplayText()
{
   int score = getCurrentScore();
   int level = getCurrentGameLevel();
   unsigned int bonus = getCurrentLevelBonus();
   int livesLeft = getNumberOfLivesThePlayerHasLeft();

   // Next, create a string from your statistics, of the form:
   // Score: 0000100  Level: 03  Lives:  3  Health:  70%  Ammo: 216  Bonus:   34

   string s = someFunctionToFormatThingsNicely(score, level, lives,
                                               health, ammo, bonus);

   // Finally, update the display text at the top of the screen with your
   // newly created stats
   setGameStatText(s);   // calls our provided GameWorld::setGameStatText
}
```

Your status line must meet the following requirements:

1. Each field's label is followed by a colon and one space.
2. Each field's value after the colon and space must be exactly as wide as shown in the example above:
   a. The Score field must be 7 digits long, with leading zeros.
   b. The Level field must be 2 digits long, with leading zeroes.
   c. The Lives field must be 2 digits long, with leading spaces (e.g., "_ 2", where _ in this sentence represents a space).

18

d. The Health field must be 3 digits long and display the player's health percentage (not hit points!), with leading spaces, and be followed by a percent sign (e.g., "_70%").

e. The Ammo field should be 3 digits long, with leading spaces (e.g., "_ 24").

f. The Bonus field must be 4 digits long, with leading spaces (e.g., "_924").

3. Each statistic must be separated from the previous statistic by two spaces. For example, between the "0000100" of the score and the "L" in "Level" there must be exactly two spaces.

You may find the Stringstreams writeup on the class web site to be helpful.

cleanUp() Details

When your *cleanUp()* method is called by our game framework, it means that the player lost a life (e.g., their hit points reached zero due to being shot) or has completed the current level. In this case, every actor in the entire maze (the player and every RageBot, goodie, crystal, marble, wall, peas, the exit, etc.) must be deleted and removed from the StudentWorld's container of active objects, resulting in an empty maze. If the user has more lives left, our provided code will subsequently call your *init()* method to reload and repopulate the maze and the level will then continue from scratch with a brand new set of actors.

You must not call the *cleanUp()* method yourself when the player dies. Instead, this method will be called by our code.

The Level Class and Level Data File

As mentioned, every level of Marble Madness has a different maze. The maze layout for each level is stored in a data file, with the file *level00.txt* holding the details for the first level's maze, *level01.txt* holding the details for the second level's maze, etc.

Here's an example maze data file (you can modify our maze data files to create wacky new levels, or add your own new maze data files to add new levels, if you like):

level00.txt:

```
###############
#     @   v   #
#    b   b    #
# #         ###
#o#    b   h#e#
#o#h   b    #*#
#a#    b   h#a#
#*#h   b    #o#
#a#    b   h#o#
###h   b    # #
#         2   #
#           b #
#######o#######
#1     x   r 2#
###############
```

As you can see, the data file contains a 15x15 grid of different characters that represent the different actors in the level. Valid characters for your maze data file are:

The **@** character specifies the location of the player's avatar when starting a level. The player's avatar should also restart at this location if the player dies and must replay the current level.

The **#** character represents a wall. The perimeter of each maze MUST be surrounded completely by Walls.

The **h** character represents a Horizontal RageBot, specifiying that a RageBot that moves only horizontally starts at this location in the maze when the player starts or replays the current level.

The **v** character represents a Vertical RageBot, specifying that a RageBot that moves only vertically starts at this location in the maze when the player starts or replays the current level.

The **1** character represents a ThiefBot factory that manufactures ThiefBots.

The **2** character represents a Mean ThiefBot factory that manufactures Mean ThiefBots.

The ***** character represents a crystal that the player needs to pick up to complete the level.

The **e** character represents an extra life goodie that grants the player an extra life (but does NOT restore the player's current health!).

The **r** character represents a restore health goodie that restores the player's hit points to 100%.

The **a** character represents an ammo goodie that gives the player 20 additional peas.

The **x** character represents the level's exit. The level's exit will be visible/active only after the player has gathered all of the crystals on the level.

The **b** character represents a marble.

The **o** (lower-case o) character represents a pit in the floor.

All **space** characters represent locations where the player's avatar and robots may walk within the maze.

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 how you might use this class:

```
#include "Level.h"    // required to use our provided class

void StudentWorld::someFunc()
{
        Level lev(assetPath());

        Level::LoadResult result = lev.loadLevel("level00.txt");
        if (result == Level::load_fail_file_not_found)
                cerr << "Could not find level00.txt data file\n";
        else if (result == Level::load_fail_bad_format)
                cerr << "Your level was improperly formatted\n";
        else if (result == Level::load_success)
        {
                cerr << "Successfully loaded level\n";

                Level::MazeEntry ge = lev.getContentsOf(5,10);       // x=5, y=10
                switch (ge)
                {
                        case Level::empty:
                                cout << "5,10 is empty\n";
                                break;
                        case Level::exit:
                                cout << "5,10 is where the exit is\n";
                                break;
                        case Level::player:
                                cout << "5,10 is where the player starts\n";
                                break;
                        case Level::horiz_ragebot:
                                cout << "5,10 starts with a horiz. RageBot\n";
                                break;
                        case Level::vert_ragebot:
                                cout << "5,10 starts with a vertical RageBot\n";
                                break;
                        case Level::thiefbot_factory:
                                cout << "5,10 holds a ThiefBot factory\n";
                                break;
                        case Level::enraged_thiefbot_factory:
                                cout << "5,10 holds an enraged ThiefBot factory\n";
                                break;
                        case Level::wall:
```

```
                            cout << "Location 5,10 holds a wall\n";
                            break;
                }
        }
}
```

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 Marble Madness game has a number of different game objects, including:

- The player's Avatar
- RageBots (Horizontal and Vertical varieties)
- ThiefBots
- Mean ThiefBots
- Factories (for Regular and Mean ThiefBots)
- Peas (that can be shot by both the player and robots)
- Exits
- Walls
- Marbles
- Pits
- Crystals
- Extra Life Goodies
- Restore Health Goodies
- Ammo Goodies

Each of these game objects can occupy the maze and interact with other game objects within the maze.

Now of course, many of your game objects will share things in common – for instance, every one of the objects in the game (RageBots, the player, walls, marbles, etc.) has x,y coordinates. Many game objects have the ability to perform an action (e.g., move or shoot) during each tick of the game. Many of them can potentially be attacked (e.g., the player, robots and marbles can be attacked by peas, pits can be filled with and "attacked" by marbles, etc.) and could "die" during a tick. All of them need some attribute that indicates whether or not they are still alive or they died during the current tick, etc.

It is therefore your job to determine the commonalities between your different game objects and make sure to factor out common behaviors and traits and move these into appropriate base classes, rather than duplicate these items across your derived classes – this is in fact 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 must never

duplicate code or data member – 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 so-called *code smell*, a weakness in a design that often leads to bugs, inconsistencies, code bloat, etc.

Hint: When you notice this specification repeating the same text nearly identically in the following sections (e.g., in the Extra Life Goodie section and the Restore Health Goodie section, or in the RageBot and ThiefBots sections) you must make sure to identify common behaviors and move these into proper base classes. NEVER duplicate behaviors across classes that can be moved into a base class!

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 ThiefBot: public Actor
{
public:
        …
};

class MeanThiefBot: public ThiefBot
{
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 your classes from our *GraphObject* base class, then you won't see anything displayed on the screen! ☺

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

```
GraphObject(int imageID, int startX, int startY,
                                    Direction startDirection = none);
void setVisible(bool shouldIDisplay);
double getX() const;
double getY() const;
void moveTo(double x, double y);
Direction getDirection() const;  // Directions: none, up, down, left, right
void setDirection(Direction d);  // Directions: none, up, down, left, right
```

You may use any of 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 not redefine any of these methods in your derived classes since they are not defined as virtual in our base class.

GraphObject(int imageID, int startX, int startY, Direction startDirection) is the constructor for a new *GraphObject*. When you construct a new *GraphObject*, you must specify an image ID that indicates how the *GraphObject* should be displayed on screen (e.g., as a RageBot, a Player, a wall, etc.). You must also specify the initial x,y location of the object. The x value may range from 0 to VIEW_WIDTH-1 inclusive, and the y value may range from 0 to VIEW_HEIGHT-1 inclusive. Notice that you pass the coordinates as x,y (i.e., column, row starting from bottom left, and *not* row, column). For those objects for which the concept of a direction doesn't apply (e.g., walls or crystals), you may leave off the final Direction argument or may specify *none*; For those objects for which a direction applies (i.e., the player, robots, peas), you must specify the initial direction the object is facing, (i.e., *left*, *right*, *up*, or *down* – these constants are defined in the *GraphObject.h* file).

One of the following IDs, found in *GameConstants.h*, must be passed in for the imageID value:

```
IID_PLAYER
IID_RAGEBOT
IID_THIEFBOT
IID_MEAN_THIEFBOT
IID_ROBOT_FACTORY
IID_PEA
IID_EXIT
IID_WALL
IID_MARBLE
IID_PIT
IID_CRYSTAL
IID_RESTORE_HEALTH
IID_EXTRA_LIFE
IID_AMMO
```

New *GraphObjects* start out invisible and are **NOT** displayed on the screen until the programmer calls the *setVisible()* method with a value of true for the parameter.

setVisible(bool shouldIDisplay) is used to tell our graphical system whether or not to display a particular *GraphObject* on the screen. If you call *setVisible(true)* on a *GraphObject*, then your object will be displayed on screen automatically by our framework (e.g., a RageBot image will be drawn to the screen at the *GraphObject's* specified x,y coordinates if the object's Image ID is IID_RAGEBOT). If you call *setVisible(false)* then your GraphObject will not be displayed on the screen. When you create a new game object, always remember to call the *setVisible()* method with a value of *true* or the actor won't display on screen!

getX() and *getY()* are used to determine a GraphObject's current location in the maze. Since each GraphObject maintains its x,y location, this means that your derived classes

MUST NOT also have x,y member variables, but instead use these fucntions and *moveTo()* from the GraphObject base class.

moveTo(int x, int y) is used to update the location of a GraphObject within the maze. For example, if a RageBot's movement logic dictates that it should move to the right, you could do the following:

```
moveTo(getX()+1, y);       // move one square to the right
```

You must use the *moveTo()* method to adjust the location of a game object if you want that object to be properly animated. As with the GraphObject constructor, note that the order of the parameters to moveTo is x,y (col,row) and NOT y,x (row,col).

getDirection() is used to determine the direction a GraphObject is facing. For example, a pea fired by a robot must travel in the direction the robot is facing, so you can use this method to learn that direction.

setDirection(Direction d) is used to change the direction a GraphObject is facing. For example, when the user presses the up arrow, you can use this method to cause the player's avatar to be displayed facing up, as well as causing any peas the player fires to travel upward.

The player

Here are the requirements you must meet when implementing the player's Avatar class.

What the Avatar Must Do When It Is Created

When it is first created:

1. The Avatar object must have an image ID of IID_PLAYER.
2. The player must always start at the proper location as specified by the current level's data file. Hint: Since your *StudentWorld*'s *init()* function loads the level, it knows this x,y location to pass when constructing the Avatar object.
3. The player, in its initial state:
    a. Has 20 hit points.
    b. Has 20 peas.
    c. Faces right.

In addition to any other initialization that you decide to do in your Player class, a Player object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What the Avatar Must Do During a Tick

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

1. The Avatar must check to see if it is currently alive. If not, then the Avatar's *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, the *doSomething()* method must check to see if the user pressed a key (the section below shows how to check this). If the user pressed a key:
    a. If the user pressed the Escape key, the user is asking to abort the current level. In this case, the Avatar object should set itself to dead. The code in the *StudentWorld* class should detect that the player has died and address this appropriately (e.g., replay the level from scratch, or end the game).
    b. If the user pressed the space bar, then if the Avatar has any peas, the it will fire a pea, which reduces their pea count by 1. To fire a pea, a new pea object must be added at the square immediately in front of the player's avatar, facing the same direction as the avatar. For example, if the Avatar is at x=10,y=7 facing upward, then the pea would be created at location x=10, y=8 facing upward. When the Avatar fires a pea, it must play the SOUND_PLAYER_FIRE sound effect (see the *StudentWorld* section of this document for details on how to play a sound).

       Hint: When you create a new pea object in the proper location and facing the proper direction, give it to the StudentWorld to manage (e.g., animate) along with the other game objects.

    c. If the user asks to move up, down, left or right by pressing a directional key, then the Avatar's direction should be adjusted to the indicated direction (e.g., if the Avatar were facing upward, and the user hit the right arrow key, the player's direction should be adjusted to *right*). Then, the Avatar will try to move to the adjacent square in the direction it is facing if that square does not contain an obstruction: a marble that cannot be pushed, a wall, a pit, a robot, or a robot factory. If the player can move, it must update its location with the *GraphObject* class's *moveTo()* method.

       For information on how and when marbles can be pushed, see the *Marble* section of this document. A marble can be pushed only in the direction the player is facing, and only if the square adjacent to the marble in that direction is empty or has a pit. If the player tries to move in the direction of an adjacent marble that can be pushed, it calls a *push* method defined in the marble class to cause the marble's position to be adjusted, and the player updates its location to the square formerly occupied by the marble.

<u>What the player Must Do When It Is Attacked</u>

When the Avatar is attacked (i.e., a pea collides with him/her), the Avatar's hit points must be decremented by 2 points. If the Avatar still has hit points after this, then the game must play an impact sound effect: SOUND_PLAYER_IMPACT; otherwise, the game must set the Avatar's state to dead and play a death sound effect: SOUND_PLAYER_DIE.

<u>Getting Input From the User</u>

Since Marble Madness is a *real-time* game, you can't use the typical *getline* or *cin* approach to get a user's key press within the player's *doSomething()* method— that would stop your program and wait for the user to type something and then hit the Enter key. This would make the game awkward to play, requiring the user to hit a directional key then hit Enter, then hit a directional key, then hit Enter, etc. Instead of this approach, you will use a function called *getKey()* that we provide in our *GameWorld* class (from which your StudentWorld class is derived) to get input from the user[1]. This function rapidly checks to see if the user has hit a key. If so, the function returns true and the int variable passed to it is set to the code for the key. Otherwise, the function immediately returns false, meaning that no key was hit. This function could be used as follows:

```
void Player::doSomething()
{
    ...
    int ch;
    if (getWorld()->getKey(ch))
    {
         // user hit a key this tick!
       switch (ch)
       {
         case KEY_PRESS_LEFT:
           ... move avatar to the left ...;
           break;
         case KEY_PRESS_RIGHT:
           ... move avatar to the right ...;
           break;
         case KEY_PRESS_SPACE:
           ... add a pea in the square in front of the avatar...;
           break;

        // etc…
       }
    }
    ...
}
```

Wall

---

[1] Hint: Since your Avatar class will need to access the *getKey()* method in the *GameWorld* class (which is the base class for your *StudentWorld* class), your Avatar class (or more likely, one of its base classes) will need a way to obtain a pointer to the *StudentWorld* object it's playing in. If you look at our code example, you'll see how the player's *doSomething()* method first gets a pointer to its world via a call to *getWorld()* (a method in one of its base classes that returns a pointer to a *StudentWorld*), and then uses this pointer to call the *getKey()* method.

Walls don't really do much. They just sit still in place. Here are the requirements you must meet when implementing the Wall class.

<u>What a Wall Must Do When It Is Created</u>

When it is first created:

1. A wall object must have an image ID of IID_WALL.
2. A wall must always start at the proper location as specified by the current level's data file.
3. A wall has no direction (Hint: Its ancestor *GraphObject* base object always has the direction *none*).

In addition to any other initialization that you decide to do in your Wall class, a wall object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

<u>What a Wall Must Do During a Tick</u>

A wall must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something, the wall must do nothing. After all, it's just a wall! (What would you think it would do?)

<u>What a Wall Must Do When It Is Attacked</u>

When a wall is attacked (i.e., a pea collides with it), nothing happens to the wall.

Marble

You must create a class to represent a giant marble. Marbles may be pushed around by the player so long as there's nothing in the way. Here are the requirements you must meet when implementing the Marble class.

<u>What a Marble Must Do When It Is Created</u>

When it is first created:

1. A marble object must have an image ID of IID_MARBLE.
2. A marble must always start at the proper location as specified by the current level's data file.
3. A marble starts out with 10 hit points.
4. A marble has no direction.

In addition to any other initialization that you decide to do in your Marble class, a marble object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What a Marble Must Do During a Tick

Each time a marble is asked to do something (during a tick), it must do nothing. After all, it's a marble.

What a Marble Must Do When It Is Attacked

When a marble is attacked (i.e., a pea collides with it), it loses 2 hit points. When its hit points reach zero, it must "die" and be removed from the game. In its place will be an empty spot.

What a Marble Must Do When It Is Pushed

When a marble is pushed by the player in a particular direction:

1. It must check to see if it can be moved in the specified direction (see the rules about where marbles can be pushed below)
2. If the marble's destination is an empty spot or a pit, then the marble should move itself to that spot when pushed (the pit will take care of swallowing the marble – see the pit section for more details).
3. Otherwise, the marble can't be moved.

Rules about How/When Marbles Can Be Pushed

For purposes of explanation, assume @ represents the player, who is trying to move rightward. Further let's assume M represents a marble, # represents a wall, O represents a pit, and _ represents an empty square, and ? represents all other objects (e.g., Goodies, robots, Crystals, the exit (regardless of whether it is hidden or revealed), etc.).

Here are several scenarios and their results:

Initial State	Final State	Notes
@M_	_@M	The player moved right, pushing the marble right into the far open slot.
@MO	_@_	The player moved right, pushing the marble into the pit at the far-right, which was then filled and replaced by an empty square
@MM	@MM	The player can't move right since there's no open spot to push the middle marble (the far-right marble is in the way!).
@M#	@M#	The player can't move right since there's no open spot to push the marble (there's a wall in the way).

@M?	@M?	The player can't move right since there's no open spot to push the marble (there's a robot, factory, goodie, exit, etc. in the way).

Pea

You must create a class to represent a pea. Here are the requirements you must meet when implementing the Pea class.

What a Pea Must Do When It Is Created

When it is first created:

1. A pea object must have an image ID of IID_PEA.
2. A pea must have its x,y location specified for it – the player or robot that fires the pea must pass in this x,y location when constructing a pea object.
3. A pea must have its direction specified for it – the player or robot that fires the pea must pass in this direction when constructing the pea object.

In addition to any other initialization that you decide to do in your Pea class, a pea object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What a Pea Must Do During a Tick

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

1. The pea must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, the pea must check to see if any other objects are on its current square:
    a. If the pea is on the same square as a marble, a robot, or the player, then the pea must:
        i. Damage the object appropriately (by causing the object to lose 2 hit points) – each damaged object can then deal with this damage in its own unique way (this may result in the damaged object dying/disappearing, a sound effect being played, the user possibly getting points, etc.) Hint: The pea can tell the object that it has been damaged by calling a method that object has (presumably named *damage* or something similar) and the target object can then determine what impact, if any, this will have.
        ii. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).

iii.   Do nothing else during the current tick.
   b. If the pea is on the same square as a wall or a robot factory, it will simply hit the wall or factory and do no damage. In this case, the pea must set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick). The pea must then do nothing more during the current tick. Note: If a pea finds itself on a square with both a robot *and* a factory, then the pea must damage the robot.
   c. If the pea is on the same square as any other game object (e.g., another Pea, the exit, a pit, a goodie, etc.) or on an empty square by itself, it does not interact with the other object in any way. Continue with step #3.
3. The pea must move itself one square in its initially-specified direction.
4. Finally, after the pea has moved itself, the pea must check to see if there are objects are on its *new* square where it just moved, using the same algorithm described in step #2 above. If so, it must perform the same behavior as described in step #2 (e.g., damage the object, etc.), but does not move any further during this tick.

What a Pea Must Do When It Is Attacked

Peas can't be attacked, silly. Peas will pass right through other peas.

Pit

You must create a class to represent a pit in the ground. When a marble is moved onto the same square as a pit, the pit will be filled in by swallowing up the marble. Here are the requirements you must meet when implementing the Pit class.

What a Pit Must Do When It Is Created

When it is first created:

1. A pit object must have an image ID of IID_PIT.
2. A pit must always start at the proper location as specified by the current level's data file.
3. A pit has no direction.

In addition to any other initialization that you decide to do in your Pit class, a pit object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What a Pit Must Do During a Tick

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

1. The pit must check to see if it is alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if a marble is on the same square as the pit, then the pit must:
    a. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    b. Set the marble's state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).

This results in the pit swallowing up the marble, and both disappearing.

<u>What a Pit Must Do When It Is Attacked</u>

Pits can't be attacked. Peas will pass right over them.

Crystal

You must create a class to represent a crystal. When the player picks up a Crystal (by moving onto the same square as it), they get 50 points. Here are the requirements you must meet when implementing the crystal class.

<u>What a Crystal Must Do When It Is Created</u>

When it is first created:

1. A crystal object must have an image ID of IID_CRYSTAL.
2. A crystal must always start at the proper location as specified by the current level's data file.
3. A crystal has no direction.

In addition to any other initialization that you decide to do in your Crystal class, a crystal object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

<u>What a Crystal Must Do During a Tick</u>

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

1. The crystal must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if the player is on the same square as the crystal, then the crystal must:
    a. Inform the *StudentWorld* object that the user is to receive 50 more points. It can do this by using *StudentWorld's increaseScore()* method (inherited from *GameWorld*).
    b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).

c. Play a sound effect to indicate that the player picked up the crystal: SOUND_GOT_GOODIE.

What a Crystal Must Do When It Is Attacked

Crystals can't be attacked. Peas will pass right over them.

The Exit

You must create a class to represent the exit, the doorway the avatar must step on to complete the current level, but only after collecting all the crystals on the level. Here are the requirements you must meet when implementing the exit class.

What the Exit Must Do When It Is Created

When it is first created:

1. The exit object must have an image ID of IID_EXIT.
2. The exit must always start at the proper location as specified by the current level's data file.
3. The exit has no direction.
4. The exit must start out *invisible* when it is created, since it is revealed to the player only after the player has collected all of the crystals on the level. At that point, the exit object must be made visible, and then the player can complete the level by stepping on it.

What the Exit Must Do During a Tick

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

1. If the player is currently on the same square as the exit AND the exit is visible (because the player has collected all of the crystals on the level), then the exit must:
   a. Play a sound indicating that the player finished the level: SOUND_FINISHED_LEVEL.
   b. Inform the *StudentWorld* object that the user is to receive 2000 more points for using the exit.
   c. Inform the *StudentWorld* object somehow that the player has completed the current level.
   d. If there are any bonus points left, the player must receive the bonus points for completing the level quickly. It's your choice whether the *StudentWorld* object or the exit object grants the points.

What the Exit Must Do When It Is Attacked

The Exit can't be attacked. Peas will pass right over it.

<u>What the Exit Must Do When It Is Revealed</u>

If the player collects all of the crystals on the current level of the game, then the game must make the exit on that level visible, allowing it to be used by the player to exit the current level and advance to the next level.

You could either have your StudentWorld's *move()* method detect whether all the crystals have been collected, and if so, make the exit object visible, or you could have the exit object's *doSomething()* method determine whether all the crystals have been collected, and if so, reveal itself. This design decision is up to you.

When the exit is revealed, it must transition from being invisible (it starts in this state) to visible, and must play a reveal sound effect: SOUND_REVEAL_EXIT.

Note: The transition of the exit on a level from invisible to visible must happen only once, when the last remaining crystal on the current level has been picked up by the player. On the subsequent ticks until the player uses the exit, your program must not play the reveal sound effect again.

Extra Life Goodie

You must create a class to represent an extra life goodie. When the player picks up this goodie (by moving onto the same square as it), it gives the player an extra life! Here are the requirements you must meet when implementing the Extra Life Goodie class.

<u>What an Extra Life Goodie Must Do When It Is Created</u>

When it is first created:

1. The extra life goodie object must have an image ID of IID_EXTRA_LIFE.
2. An extra life goodie must always start at the proper location as specified by the current level's data file.
3. Extra life goodies have no direction.

In addition to any other initialization that you decide to do in your Extra Life Goodie class, an extra life goodie object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

<u>What an Extra Life Goodie Must Do During a Tick</u>

Each time an extra life goodie is asked to do something (during a tick):

1. The extra life goodie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if the player is on the same square as the extra life goodie, then the extra life goodie must:
   a. Inform the *StudentWorld* object that the user is to receive 1000 more points.
   b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   c. Play a sound effect to indicate that the player picked up the goodie: SOUND_GOT_GOODIE.
   d. Inform the *StudentWorld* object that the player is to gain one extra life.

<u>What an Extra Life Goodie Must Do When It Is Attacked</u>

Extra life goodies can't be attacked. Peas will pass right over them.

Restore Health Goodie

You must create a class to represent a restore health goodie. When the player picks up this goodie (by moving onto the same square as it), it restores the player's health. Here are the requirements you must meet when implementing the Restore Health Goodie class.

<u>What a Restore Health Goodie Must Do When It Is Created</u>

When it is first created:

1. The restore health goodie object must have an image ID of IID_RESTORE_HEALTH.
2. A restore health goodie must always start at the proper location as specified by the current level's data file.
3. Restore health goodies have no direction.

In addition to any other initialization that you decide to do in your Restore Health Goodie class, a restore health goodie object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

<u>What a Restore Health Goodie Must Do During a Tick</u>

Each time a restore health goodie is asked to do something (during a tick):

1. The restore health goodie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.

2. Otherwise, if the Avatar is on the same square as the restore health goodie, then the restore health goodie must:
   a. Inform the *StudentWorld* object that the user is to receive 500 more points.
   b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   c. Play a sound effect to indicate that the player picked up the Goodie: SOUND_GOT_GOODIE.
   d. Inform the Avatar object that the player is to restore itself to full health (i.e., 20 hit points, the initial value).

What a Restore Health Goodie Must Do When It Is Attacked

Restore health goodies can't be attacked. Peas will pass right over them.

Ammo Goodie

You must create a class to represent an ammo goodie. When the player picks up this goodie (by moving onto the same square as it), it adds 20 peas to the player pea supply. Here are the requirements you must meet when implementing the Ammo Goodie class.

What an Ammo Goodie Must Do When It Is Created

When it is first created:

1. The ammo goodie object must have an image ID of IID_AMMO.
2. An ammo goodie must always start at the proper location as specified by the current level's data file.
3. Ammo goodies have no direction.

In addition to any other initialization that you decide to do in your Ammo Goodie class, an ammo goodie object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What an Ammo Goodie Must Do During a Tick

Each time an ammo goodie is asked to do something (during a tick):

1. The ammo goodie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if the player is on the same square as the ammo goodie, then the ammo goodie must:
   a. Inform the *StudentWorld* object that the user is to receive 100 more points.
   b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).

  c. Play a sound effect to indicate that the player picked up the goodie: SOUND_GOT_GOODIE.

  d. Inform the player that the player is to add 20 peas.

<u>What an Ammo Goodie Must Do When It Is Attacked</u>

Ammo goodies can't be attacked. Peas will pass right over them.

RageBot

You must create a class to represent a RageBot. Here are the requirements you must meet when implementing the RageBot class.

<u>What a RageBot Must Do When It Is Created</u>

When it is first created:

1. The RageBot object must have an image ID of IID_RAGEBOT.
2. A RageBot must always start at the proper location as specified by the current level's data file.
3. A RageBot must start out facing either right or down, depending on whether it's a horizontal or vertical RageBot.
4. A RageBot must start out with 10 hit points.

In addition to any other initialization that you decide to do in your RageBot class, a RageBot object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

A RageBot, unlike the player, doesn't necessarily get to take an action during every tick of the game. (This is to make the game easier to play, since if a RageBot moved once every tick, it would move much faster than the typical user can think and hit the keys on the keyboard.) A RageBot must therefore compute a value indicating how frequently it's allowed to take an action. This value is to be computed as follows:

```
int ticks = (28 - levelNumber) / 4;  // levelNumber is the current
                                     // level number (0, 1, 2, etc.)
if (ticks < 3)
   ticks = 3;  // no RageBot moves more frequently than this
```

If the value of *ticks* is 5, for example, then the RageBot must "rest" for 4 ticks, perform its normal behavior on the next tick, rest for 4 ticks, perform its behavior on the next tick, etc., performing its behavior every 5th tick.

<u>What a RageBot Must Do During a Tick</u>

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

1. The RageBot must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. If the RageBot is supposed to "rest" during the current tick, it must during the current tick other than to update its tick count.
3. Otherwise, the RageBot must determine whether it should fire its pea cannon: If the player is in the same row or column as the RageBot AND the RageBot is currently facing the player AND there are no obstacles (specifically, walls, marbles, robots, or robot factories) in the way, then the RageBot will fire a pea toward the player and then do nothing more during the current tick.

   To fire a pea, a new pea object must be added at the square immediately in front of the RageBot, facing the same direction as the RageBot. For example, if the RageBot is at x=10,y=7 facing upward, then the pea would be created at location x=10, y=8 facing upward. Every time a RageBot fires a pea, it must play the SOUND_ENEMY_FIRE sound effect.

   Hint: When you create a new pea object in the proper location and facing the proper direction, give it to your StudentWorld to manage (e.g., animate) along with the other game objects.

4. Otherwise, the RageBot will try to move to the adjacent square in the direction it is facing if that square does not contain an obstruction: the player, a marble, a wall, a pit, a robot, or a robot factory.
   a. If the square does not contain an obstruction, the RageBot will move to it.
   b. Otherwise, the RageBot <u>will not move</u>, but instead reverse the direction it's facing (left ←→ right or up←→down).

<u>What a RageBot Must Do When It Is Attacked</u>

A RageBot should have a method named *damage* or something similar that a pea object can call to inform the RageBot that it has been damaged. When a RageBot is damaged:
1. If its hit points have not reached zero, the game must play the SOUND_ROBOT_IMPACT sound effect.
2. Otherwise, the RageBot has been killed, so it must:
   a. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   b. Play a sound indicating that it has died: SOUND_ROBOT_DIE.
   c. Inform the *StudentWorld* object that the user is to receive 100 more points.

ThiefBot

You must create a class to represent a ThiefBot. Here are the requirements you must meet when implementing the ThiefBot class.

What a ThiefBot Must Do When It Is Created

When it is first created:

1. The ThiefBot object must have an image ID of IID_THIEFBOT.
2. A ThiefBot must always start at the proper location as specified by the current level's data file.
3. A ThiefBot must start out facing right.
4. A ThiefBot must start out with 5 hit points.
5. A ThiefBot selects a random integer from 1 to 6 inclusive, which we'll call *distanceBeforeTurning*. This is how far the ThiefBot will move in a straight line (if it can) before turning.

In addition to any other initialization that you decide to do in your ThiefBot class, a ThiefBot object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

A ThiefBot, unlike the player, doesn't necessarily get to take an action during every tick of the game. (This is to make the game easier to play, since if a ThiefBot moved once every tick, it would move much faster than the typical user can think and hit the keys on the keyboard.) A ThiefBot must therefore compute a value indicating how frequently it's allowed to take an action. This value is to be computed as follows:

```
int ticks = (28 - levelNumber) / 4;  // levelNumber is the current
                                     // level number (0, 1, 2, etc.)
if (ticks < 3)
   ticks = 3;  // no ThiefBot moves more frequently than this
```

If the value of *ticks* is 5, for example, then the ThiefBot must "rest" for 4 ticks, perform its normal behavior on the next tick, rest for 4 ticks, perform its behavior on the next tick, etc., performing its behavior every 5th tick.

What a ThiefBot Must Do During a Tick

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

1. The ThiefBot must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. If the ThiefBot is supposed to "rest" during the current tick, it must do nothing during the current tick other than to update its tick count.
3. Otherwise, if the ThiefBot is on the same square as a goodie, and has not yet ever picked up a goodie, then there is a 1 in 10 chance that it will pick up that goodie. If it does:
   a. That Goodie must no longer appear on the screen and must not be able to be picked up by the player until the ThiefBot is killed.

b. The ThiefBot must play a "munching" sound: SOUND_ROBOT_MUNCH.
c. The ThiefBot must do nothing more during the current tick.

Hint: You can either (1) make the goodie invisible and unable to be picked up while the ThiefBot is carrying it, undoing that when the ThiefBot is killed, or (2) have the ThiefBot remember the kind of goodie it picked up, kill the goodie object, and when the ThiefBot is killed, create a new goodie object of that type. When the ThiefBot is killed, the goodie must appear at the ThiefBot's location.

4. Otherwise, if the ThiefBot has not yet moved *distanceBeforeTurning* squares in its current direction, then it will try to move to the adjacent square in the direction it is facing if that square does not contain an obstruction: the player, a marble, a wall, a pit, a robot, a robot Factory, or the player. If the ThiefBot can move, it must update its location to that square and do nothing more during the curent tick.
5. Otherwise, the ThiefBot has either moved *distanceBeforeTurning* squares in a straight line or encountered an obstruction. In this case, the ThiefBot must:
   a. Select a random integer from 1 to 6 inclusive to be the new value of *distanceBeforeTurning*.
   b. Select a random direction, which we'll call *d*. It's OK if this direction happens to be the same as the ThiefBot's current direction.
   c. Starting by considering the direction *d*, repeat the following until it has either successfully moved or cannot move because of obstructions in all four directions:
      i. Determine whether there is an obstruction in the adjacent square in the direction under consideration. If there is none, the ThiefBot must set its direction to the direction under consideration, move to that square, and then do nothing more during the current tick.
      ii. If there is an obstruction, the ThiefBot must consider a direction it has not already considered during this loop, and repeat step i.
   If there is an obstruction in all four directions, then the ThiefBot must set its current direction to *d* and do nothing more during the current tick.

What a ThiefBot Must Do When It Is Attacked

A ThiefBot should have a method named *damage* or something similar that a pea object can call to inform the ThiefBot that it has been damaged. When a ThiefBot is damaged:
1. If its hit points have not reached zero, the game must play the SOUND_ROBOT_IMPACT sound effect.
2. Otherwise, the ThiefBot has been killed, so it must:
   a. If it had picked up a goodie, make that goodie appear on the screen and be able to be picked up by the player.
   b. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   c. Play a sound indicating that it has died: SOUND_ROBOT_DIE.
   d. Inform the *StudentWorld* object that the user is to receive 10 more points.

Mean ThiefBot

You must create a class to represent a Mean ThiefBot. Here are the requirements you must meet when implementing the Mean ThiefBot class.

What a Mean ThiefBot Must Do When It Is Created

When it is first created:

1. The Mean ThiefBot object must have an image ID of IID_MEAN_THIEFBOT.
2. A Mean ThiefBot must always start at the proper location as specified by the current level's data file.
3. A Mean ThiefBot must start out facing right.
4. A Mean ThiefBot must start out with 8 hit points.
5. A Mean ThiefBot selects a random integer from 1 to 6 inclusive, which we'll call *distanceBeforeTurning*. This is how far the Mean ThiefBot will move in a straight line (if it can) before turning.

In addition to any other initialization that you decide to do in your Mean ThiefBot class, a Mean ThiefBot object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

A Mean ThiefBot, unlike the player, doesn't necessarily get to take an action during every tick of the game. (This is to make the game easier to play, since if a Mean ThiefBot moved once every tick, it would move much faster than the typical user can think and hit the keys on the keyboard.) A Mean ThiefBot must therefore compute a value indicating how frequently it's allowed to take an action. This value is to be computed as follows:

```
int ticks = (28 - levelNumber) / 4;  // levelNumber is the current
                                      // level number (0, 1, 2, etc.)
if (ticks < 3)
    ticks = 3;  // no Mean ThiefBot moves more frequently than this
```

If the value of *ticks* is 5, for example, then the Mean ThiefBot must "rest" for 4 ticks, perform its normal behavior on the next tick, rest for 4 ticks, perform its behavior on the next tick, etc., performing its behavior every 5th tick.

What a Mean ThiefBot Must Do During a Tick

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

1. The Mean ThiefBot must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.

2. If the Mean ThiefBot is supposed to "rest" during the current tick, it must do nothing during the current tick other than to update its tick count.
3. Otherwise, the Mean ThiefBot must determine whether it should fire its pea cannon: If the player is in the same row or column as the Mean ThiefBot AND the Mean ThiefBot is currently facing the player AND there are no obstacles (specifically, Walls, Marbles, robots, or robot factories) in the way, then the Mean ThiefBot will fire a pea toward the player and then do nothing more during the current tick.

   To fire a pea, a new pea object must be added at the square immediately in front of the Mean ThiefBot, facing the same direction as the Mean ThiefBot. For example, if the Mean ThiefBot is at x=10,y=7 facing upward, then the Pea would be created at location x=10, y=8 facing upward. Every time a Mean ThiefBot fires a pea, it must play the SOUND_ENEMY_FIRE sound effect.

   Hint: When you create a new pea object in the proper location and facing the proper direction, give it to your StudentWorld to manage (e.g., animate) along with the other game objects.

4. Otherwise, if the Mean ThiefBot is on the same square as a goodie, and has not yet ever picked up a goodie, then there is a 1 in 10 chance that it will pick up that Goodie. If it does:
   a. That Goodie must no longer appear on the screen and must not be able to be picked up by the player until the Mean ThiefBot is killed.
   b. The Mean ThiefBot must play a "munching" sound: SOUND_ROBOT_MUNCH.
   c. The Mean ThiefBot must do nothing more during the current tick.

   Hint: You can either (1) make the Goodie invisible and unable to be picked up while the Mean ThiefBot is carrying it, undoing that when the Mean ThiefBot is killed, or (2) have the Mean ThiefBot remember the kind of goodie it picked up, kill the goodie object, and when the Mean ThiefBot is killed, create a new Goodie object of that type. When the Mean ThiefBot is killed, the goodie must appear at the Mean ThiefBot's location.

5. Otherwise, if the Mean ThiefBot has not yet moved *distanceBeforeTurning* squares in its current direction, then it will try to move to the adjacent square in the direction it is facing if that square does not contain an obstruction: the player, a marble, a wall, a pit, a robot, a robot factory, or the player. If the Mean ThiefBot can move, it must update its location to that square and do nothing more during the current tick.
6. Otherwise, the Mean ThiefBot has either moved *distanceBeforeTurning* squares in a straight line or encountered an obstruction. In this case, the Mean ThiefBot must:
   a. Select a random integer from 1 to 6 inclusive to be the new value of *distanceBeforeTurning*.

b. Select a random direction, which we'll call *d*. It's OK if this direction happens to be the same as the Mean ThiefBot's current direction.

c. Starting by considering the direction *d*, repeat the following until it has either successfully moved or cannot move because of obstructions in all four directions:

  i. Determine whether there is an obstruction in the adjacent square in the direction under consideration. If there is none, the Mean ThiefBot must set its direction to the direction under consideration, move to that square, and then do nothing more during the current tick.

  ii. If there is an obstruction, the Mean ThiefBot must consider a direction it has not already considered during this loop, and repeat step i.

If there is an obstruction in all four directions, then the Mean ThiefBot must set its current direction to *d* and do nothing more during the current tick.

What a Mean ThiefBot Must Do When It Is Attacked

A Mean ThiefBot should have a method named *damage* or something similar that a pea object can call to inform the Mean ThiefBot that it has been damaged. When a Mean ThiefBot is damaged:

1. If its hit points have not reached zero, the game must play the SOUND_ROBOT_IMPACT sound effect.
2. Otherwise, the Mean ThiefBot has been killed, so it must:
   a. If it had picked up a goodie, make that goodie appear on the screen and be able to be picked up by the player.
   b. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   c. Play a sound indicating that it has died: SOUND_ROBOT_DIE.
   d. Inform the *StudentWorld* object that the user is to receive 20 more points.

ThiefBot Factory

You must create a class to represent a ThiefBot Factory. ThiefBot Factories come in two forms: one that manufactures regular ThiefBots and one that manufactures Mean ThiefBots. A ThiefBot Factory churns out new ThiefBots and adds them to the current level according to rules described below. Here are the requirements you must meet when implementing the ThiefBot class.

What a ThiefBot Factory Must Do When It Is Created

When it is first created:

1. The ThiefBot factory object must have an image ID of IID_ROBOT_FACTORY.

2. A ThiefBot factory must always start at the proper location as specified by the current level's data file.
3. A ThiefBot factory has no direction.
4. A ThiefBot factory is told whether it is to produce regular ThiefBots or Mean ThiefBots.

In addition to any other initialization that you decide to do in your ThiefBot factory class, a ThiefBot factory object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

<u>What a ThiefBot Factory Must Do During a Tick</u>

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

1. The ThiefBot factory must count how many ThiefBots of either type are present in the square region that extends from itself 3 squares up, 3 squares left through 3 squares down, 3 squares right. For example, a factory at x=10,y=8 would count all the ThiefBots and Mean ThiefBots in the region bounded by x=7,y=11; x=13,y=11; x=13,y=5; and x=7,y=5, inclusive. A factory at x=2,y=13 would count the ThiefBots and Mean ThiefBots in the region bounded by x=0,y=14; x=5,y=14; x=5,y=10; and x=0,y=10, inclusive, properly accounting for the boundaries of the maze.
2. If the count is less than 3 AND there is no ThiefBot of any type on the same square as the factory, then there is a 1 in 50 chance that during the current tick, the Factory will create a new ThiefBot of the type that factory manufactures and add it to the maze on the same square as the factory. If it creates a new ThiefBot, it must play the SOUND_ROBOT_BORN sound effect.

<u>What a ThiefBot Factory Must Do When It Is Attacked</u>

ThiefBot factory can't be attacked. Peas that smack into them do no damage to them. However, if a ThiefBot of any type is on the same square as a factory (because it was just created by the Factory), then a pea that moves onto that square damages the ThiefBot as usual.

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_RAGEBOT, IID_PLAYER, IID_WALL, 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.). 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 <u>classes of behaviors</u>:**

   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 used directly only by other methods within the same class that holds it. Make it private or protected instead.**

7. **Your StudentWorld public methods should never return a collection of the objects StudentWorld maintains or a pointer to such a collection. (Returning a**

pointer to a single object in the collection is OK.) Only StudentWorld should know about all of its game objects and where they are. If an action requires traversing StudentWorld's collection, then a StudentWorld method should do it.

Don't do this:

```
class StudentWorld
{
  public:
    vector<Actor*> getActorsThatCanBeZapped(int x, int y)
    {
        …       // create a vector with a actor pointers and return it
    }
};

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

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


    }
};
```

Do this instead:

```
class StudentWorld
{
  public:
    void zapAllZappableActors(int x, int y)
    {
      for (p = actors.begin(); p != actors.end(); p++)
        if (p->isAt(x,y) && p->isZappable())
          p->zap();
    }
};

class NastyRobot
{
  public:
    virtual void doSomething()
    {
            …
      studentWorldPtr->zapAllZappableActors(getX(), getY());
    }
};
```

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, ONLINE STORAGE, OR A PRIVATE GITHUB REPOSITORY EVERY TIME YOU MAKE A MEANINGFUL CHANGE.

WE WILL NOT ACCEPT EXCUSES THAT YOUR HARD DRIVE/COMPUTER CRASHED OR THAT YOUR CODE USED TO WORK UNTIL YOU MADE THAT ONE CHANGE (AND DON'T KNOW WHAT CAUSED IT TO BREAK).

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., `"Z:/MarbleMadness/Assets"` or `"/Users/fred/MarbleMadness/Assets"`).

To build the game, follow these steps:

For Windows

Unzip the MarbleMadness-skeleton-windows.zip archive into a folder on your hard drive. Double-click on MarbleMadness.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 macOS

Unzip the MarbleMadness-skeleton-mac.zip archive into a folder on your hard drive. Double-click on our provided MarbleMadness.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*/BuildProducts/Debug* (e.g., */Users/fred/MarbleMadness/DerivedData/MarbleMadness/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 scared to death about this project and don't know where to start. So, we're going to incentivize you to work incrementally rather than try to do everything all at once. For the first part of Project 3, your job is to build a really simple version of the Marble Madness game that implements maybe 15% of the overall project. You must program:

1. A class that can serve as the base class for all of your game's actors (e.g., the Avatar, RageBots, ThiefBots, goodies, walls, pits, crystals, etc.):
    i. It must have a simple constructor.
    ii. It must be derived from our GraphObject class.
    iii. It must have a member function named *doSomething()* that can be called to cause the actor to do something.
    iv. You may add other public/private member functions and private data members to this base class, as you see fit.
2. A wall class, derived in some way from the base class described in 1 above:
    i. It must have a simple constructor.
    ii. It must have an Image ID of IID_WALL.
    iii. It (or its base class) must make itself visible via a call to *setVisible(true);*
    iv. You may add other public/private member functions and private data members to your Wall class as you see fit, so long as you use good object-oriented programming style (e.g., you must NOT duplicate non-trivial functionality across classes).
3. A limited version of your Avatar class, derived in some way from the base class described in 1 just above (either directly derived from the base class, or derived from some other class that is somehow derived from the base class):
    i. It must have a constructor that initializes the player – see the player section for more details on where to initialize the player.

ii. It must have an Image ID of IID_PLAYER.

iii. It (or its base class) must make itself visible via a call to
setVisible(true);

iv. It must have a limited version of a *doSomething()* method that lets the
user pick a direction by hitting a directional key. If the player hits a
directional key during the current tick and the target square does not
contain a wall, it updates the player's location to the target square and
the player's direction. All this *doSomething()* method has to do is
properly adjust the player's x,y coordinates and direction, and our
graphics system will automatically animate its movement it around the
maze!

v. You may add other public/private member functions and private data
members to your Player class as you see fit, so long as you use good
object-oriented programming style (e.g., you must not duplicate
functionality across classes).

4. A limited version of the *StudentWorld* class.

i. Add any private data members to this class required to keep track of
walls as well as the Avatar object. You may ignore all other items in
the maze such as RageBots, the exit, etc. for part #1.

ii. Implement a constructor for this class that initializes your data
members.

iii. Implement a destructor for this class that frees any remaining
dynamically allocated data that has not yet been freed at the time the
StudentWorld object is destroyed.

iv. Implement the *init()* method in this class. It must create the player and
insert it into the maze at the right starting location (see the Level class
section of this document for details on the starting location). It must
also create all of the Walls and add them to the maze as specified in
the current Level's data file.

v. Implement the *move()* method in your *StudentWorld* class. During
each tick, it must ask your Avatar and walls to do something. Your
move() method need not check to see if the player has died or not; you
may assume at this point that the player cannot die. Your *move()*
method does not have to deal with any actors other than the player and
the Walls.

vi. Implement a *cleanup()* method that frees any dynamically allocated
data that was allocated during calls to the *init()* method or the *move()*
method (e.g., it should delete all your allocated Walls and the player).
Note: Your *StudentWorld* class must have both a destructor and the
cleanUp() method even though they likely do the same thing (in which
case the destructor could just call *cleanup()*).

As you implement these classes, repeatedly build your program – you'll probably start
out with lots of errors… Relax and try to remove them and get your program to run.
(Historical note: A UCLA student taking CS131 once got 1,800 compilation errors when

compiling a 900-line class project written in the Ada programming language. His name was Carey Nachenberg. Somehow he survived and has lived a happy life since then.)

Note: Your class names don't have to be exactly the ones we used in this description. For example, Avatar, PlayerAvatar, Player, or something similar would be reasonable names for what we called the Avatar class.

You'll know you're done with part 1 when your program builds and does the following: When it runs and the user hits Enter to begin playing, it displays a maze with the player in its proper starting position. If your classes work properly, you should be able to move the player around the maze using the directional keys, without the player walking through any walls.

Your Part #1 solution may actually 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.

Note, the Part #1 specification above doesn't require you to implement any RageBots, ThiefBots, marbles, pits, crystals, any goodies, or the exit (unless you want to). You may do these unmentioned items if you like but they're not required for Part 1. **However, if you add additional functionality, make sure that your Avatar, Wall, and StudentWorld classes still work properly and that your program still builds and meets the requirements stated above for Part #1!**

If you can get this simple version working, you'll have done a bunch of the hard design work. You'll probably still have to change your classes a lot to implement the full project, but you'll have done most of the hard thinking.

What to Turn In For Part #1

You must turn in your source code for the simple version of your game, which **must build without errors** 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 four files:

Actor.h	// contains base, Avatar, and Wall class declarations
	// 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

You will not be turning in any other files – we'll test your code with our versions of the 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 will not turn in a report for Part #1; we will not be evaluating Part #1 for

program comments, documentation, or test cases; all that matters for Part #1 is correct behavior for the specified subset of the requirements.

Part #2 (80%)

After you have turned in your work for Part #1 of Project 3, we will discuss one possible design for this assignment. For the rest of this project, you are welcome to continue to improve the design that you came up with for Part #1, **or you can use the design we provide**.

In Part #2, your goal is to implement a fully working version of the Marble Madness game, which adheres exactly to the functional specification provided in this document.

What to Turn In For Part #2

You must turn in your source code for your game, which **must build without errors** 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 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"` or the value of the constant `msPerTick` in main.cpp.)

You must turn in a report that contains the following:

1. A description of the control flow for the interaction of the player avatar and a goodie. 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 Mean ThiefBot doesn't work correctly yet so I treat it like a ThiefBot 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: Why does my video game run slower/faster than yours?
A: It could be your choice of data structures and algorithms, graphics cards, etc. It's OK if your game is faster or a little slower than ours. If your game runs MUCH slower, then you probably have a Big-O problem and could choose better data structures. If your game runs faster and you'd like to slow down gameplay, update line 14 in main.cpp with a larger value, like 20 or 50:

```
const int msPerTick = 10;
```

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 it not even building!

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

GOOD LUCK!