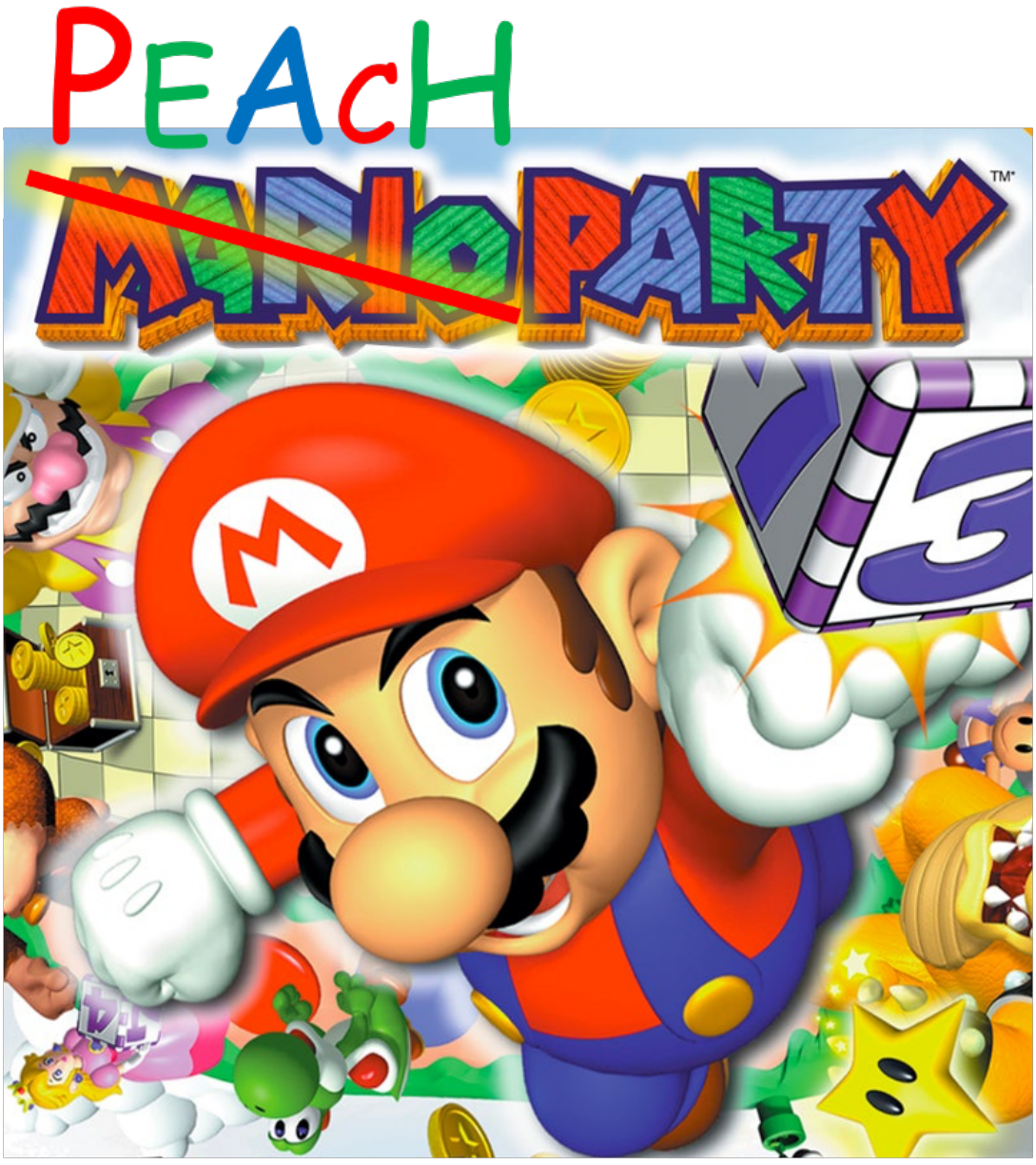
Project 3
Peach Party

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



Time due:
Part 1: 11 PM, Sunday, February 26
Part 2: Sunday, March 5

WHEN IN DOUBT ABOUT A REQUIREMENT, YOU WILL NEVER LOSE CREDIT IF YOUR SOLUTION WORKS THE SAME AS OUR POSTED SOLUTION.

SO PLEASE DO NOT ASK ABOUT ITEMS WHERE YOU CAN DETERMINE THE PROPER BEHAVIOR ON YOUR OWN FROM OUR SOLUTION!

BACK UP YOUR SOLUTION EVERY 30 MINUTES TO THE CLOUD OR A THUMB DRIVE. WE WILL NOT ACCEPT "MY COMPUTER CRASHED" EXCUSES FOR LATE WORK.

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

Thanks to Steven Bui for creating an awesome Mario Party clone from scratch!

Table of Contents

Introduction

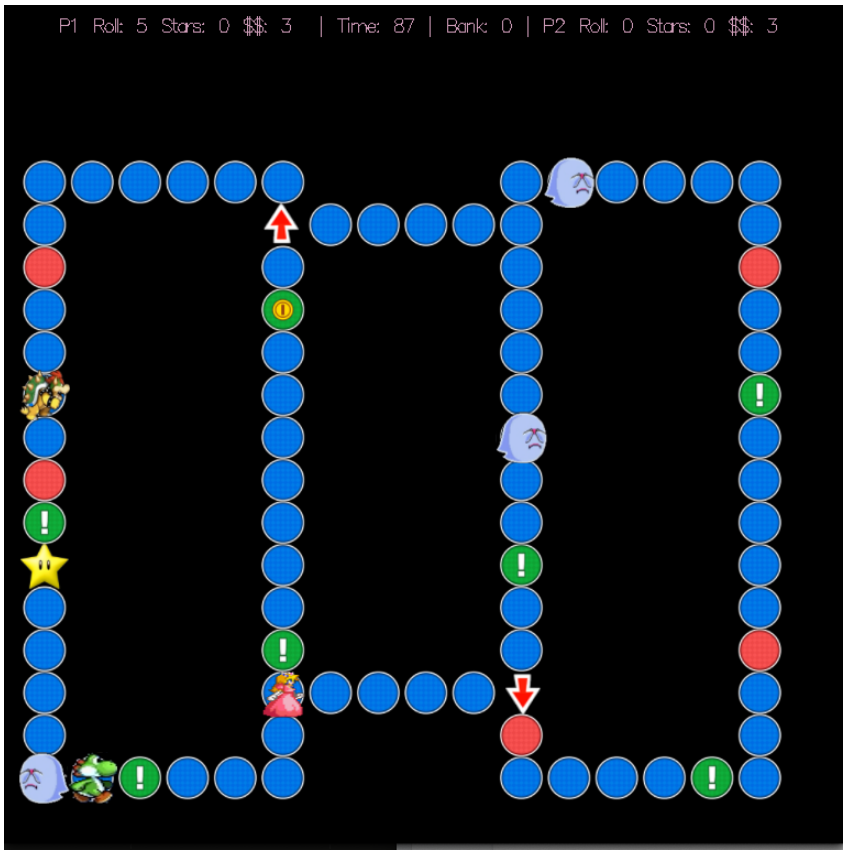
NachenGames corporate spies have learned that SmallSoft is planning to release a new video game called Peach Party - a clone of Mario Party - and would like you to program an exact copy so NachenGames can beat SmallSoft to the market. To help you, NachenGames corporate spies have managed to steal a prototype Peach Party 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.

Peach Party is a *two-player game* set in a world that resembles a traditional board game. The players must direct their avatars, Peach and Yoshi, along the squares on the game board collecting coins and stars, while trying to avoid baddies like Bowser and Boo who cause all sorts of trouble. When the game starts, the players may choose what board they want to play by pressing a key from 1 to 9; each board has a different layout and set of challenges. The winner of the game is the player who has collected the most stars and coins during 99 seconds of game play. If both players have collected the same number of stars, then the number of coins is used as a tie-breaker. If both players have the same number of stars and coins, then a winner is randomly picked.

Like a board game, Peach Party requires the players to roll a (virtual) 10-sided die to decide how far they can move along the board during their next turn. To roll a virtual die, each player hits a key (more details below). After rolling, the player that rolled then travels along the board's squares in their current direction until they've moved the appropriate number of squares, at which time they "land" on a square. Then they get to roll the die again. If a player hits a fork on the board where they could move in more than one direction, they may select one of the paths to take (e.g., up vs. down vs. left) using the keyboard, and then they will continue passing over squares until they exhaust their die roll.

Many of the squares on the Peach Party game board have special powers - some will grant or deduct coins or stars from the players when they're landed on. Others will cause the player to teleport to another square on the board, give the player Vortex projectiles to shoot at the baddies, etc. In most cases, a square will only "activate" on Peach or Yoshi if they land upon it as their final move of the current die roll. For example, if Yoshi rolls a 5, he will walk over the first four squares (which will have no impact on him), and then he will "land" on the fifth square. This fifth square will then have an opportunity to activate on Yoshi, for example granting him three coins. However, in some cases squares activate as they're walked over, even if they're not landed upon.

Here's a screenshot from the game:



Here's a screenshot of the Peach Party game. We can see a Boo - a baddie - in the bottom left corner, with Yoshi (the green fellow with brown shoes) to that Boo's right. Peach can be seen in her pink dress up and to the right of them. Bowser is hanging out in the left column of the board midway down. We also see Blue Coin Squares (which give coins), Red Coin Squares (which take coins), Star Squares which give stars, Event Squares (!) which choose from 3 random effects when players land on them, a Bank Square (green with yellow center), and Directional Squares (with arrows) which change Peach and Yoshi's direction.

The player controlling Peach uses the following keys to direct her activity:

- To roll the die: Tab key
- To choose a left-fork on the board: the 'a' key
- To choose a right-fork on the board: the 'd' key
- To choose the up-fork on the board: the 'w' key
- To choose the down-fork on the board: the 's' key
- To fire a Vortex projectile in the current direction (if Peach has one): the back quote ` key

The player controlling Yoshi uses the following keys to direct his activity:

- To roll the die: Enter key

- To choose a left-fork on the board: left arrow key
- To choose a right-fork on the board: right arrow key
- To choose the up-fork on the board: up arrow key
- To choose the down-fork on the board: down arrow key
- To fire a Vortex projectile in the current direction (if Yoshi has one): the backslash \ key

The game developer (that's you!) can also use the following commands which are built into our game framework - your code doesn't need to do anything to process them:

- The Escape key: Quits the game
- The ‘f’ key: Causes the game to run one tick at a time to help you with debugging; hit 'r' to resume full-speed play

Game Details

There are multiple possible boards you can choose to play in Peach Party, numbered board #1 through board #9. The players get to pick a board from the starting screen of the game by pressing 1-9. One gameplay begins, Peach and Yoshi have 99 seconds to move around the board and collect as many stars and coins as they can before the game is over. The player with the most stars/coins wins the game.

At the beginning of gameplay, all of the "actors" in the game are placed in their initial positions and start with their initial states as specified by the current board's data file. Peach and Yoshi both start out on a designated starting space on the game board, *facing toward the right*. The humans (aka players) controlling Peach and Yoshi must each hit a key to initiate a die roll to determine how far they're going to move, and then their *avatars* will move in their designated direction (initially right) until they exhaust their die roll or reach a fork, at which point the human player may select a direction for them to continue moving up to their die roll. When a player's avatar finishes moving the number of squares designated by their die roll, a player may hit a key to roll the die again, and again move along the board's squares in their current direction. Players may hit keys and move at the same time - there are no alternating turns in Peach World like in games like Monopoly, so players need to make sure they are quick with the keyboard to maximize their trip around the board.

The Peach Party screen is exactly 256 pixels wide by 256 pixels high. The bottom-leftmost pixel has coordinates x=0,y=0, while the upper-rightmost pixel has coordinates x=255, y=255, where x increases to the right and y increases upward toward the top of the screen. The *GameConstants.h* file we provide defines constants that represent the game’s width and height (VIEW_WIDTH and VIEW_HEIGHT), which you must use in your code instead of hard-coding the integers. Every object in the game (e.g., Peach, Yoshi, various squares, Bowser, Boo, Vortexes, etc.) will have an x coordinate in the range 0 to VIEW_WIDTH−1 inclusive, and a y coordinate in the range 0 to VIEW_HEIGHT−1 inclusive.

Each board has its layout defined in a data file, such as board01.txt or board02.txt. We provide nine example board files in the *Assets* folder (board03.txt through board09.txt are currently identical); you may modify these data files to create new and exciting boards. The boards are always 16 squares wide by 16 squares high. For more details on the format of the board data files, please see the Board Data File section.

After a player rolls the die, their avatar moves forward in the its currently-facing direction, traveling over the number of squares indicated by the die. A player may only choose a new direction for their avatar if they reach a fork on the board where there are multiple ways to go (e.g., up *and* left). If a player reaches a square on the board where they can no longer continue moving in their current direction and must turn (e.g., they're traveling right and reach the bottom-right square on the screen and need to turn left to start traveling upward on the board), their avatar will automatically follow the adjacent square on the board in the new direction (e.g., upward) without needing the player to specify the new direction by hitting a key.

Peach and Yoshi will always be in one of two states: *waiting to roll* or *walking*. A player's avatar is in the *waiting to roll* state (a) at the start of the game, and (b) when they have finished moving to satisfy their previous die roll but have not yet rolled the die again. So if Peach rolled a 4 and traveled all 4 squares but had not yet re-rolled the die, she'd be in the *waiting to roll* state. A player is in the *walking* state once they have rolled the die and have not yet moved the total number of squares indicated by their last die roll (they still have squares to travel across). So if Yoshi rolled a 4 and has traveled over 3 squares so far, he'd still be in the *walking* state. A player who is at a fork on the board and waiting to select a direction to move would also be in the *walking* state, since they have not yet exhausted their current die roll (even though the avatar is not actively moving since it's waiting for the player to pick a direction).

Some of the squares on a board perform special actions when they are stepped on by a player.

If either Peach or Yoshi *lands on*[1] one of the following types of squares:

- Blue Coin Square: Gives 3 coins to the player
- Red Coin Square: Takes 3 coins from the player
- Star Square: If the player has at least 20 coins, this square gives the user 1 additional star, and takes away 20 coins from the user
- Event Square: Randomly does one of the following:
    - Teleports the player to a random square
    - Swaps the player's position with the other player, or

[1] By "lands on" a square, we mean that the player's avatar finished on the square after moving the number of squares designated by their die roll. So if the player rolled 5 on the die, their avatar would pass over the first four squares, and "land on" the fifth square. To "land on" a square, the avatar and the square must share the same X,Y coordinates. Simply overlapping, but not sharing the exact same X,Y coordinates, is not sufficient!

    - Gives the player a Vortex projectile to shoot
- Bank Square: Gives the player all the coins stored in the central bank
- Directional Square: Forces the player to face in the direction indicated by the square (up, down, left or right); the square changes the player's direction
- Dropping Square: Takes coins or a star from the player

then the square will activate once and perform their action on that player. The square will not activate again on the same player until that player has moved off of the square and re-landed back upon it (there are no double activations).

If either Peach or Yoshi *moves onto*[2] one of the following types of squares while the player is in the *walking* state (but does not land on the square, just passes over it):

- Star Square: If the player has at least 20 coins, this square gives the user 1 additional star, and takes away 20 coins from the user (so a Star Square activates both if the player walks over it *or* lands on it)
- Directional Square: Forces the player to face in the specified direction; the square changes the player's direction (so a Directional Square activates either if the player walks over it or lands on it)
- Bank Square: Deducts 5 coins from the player and deposits them into the central bank account

then the square will activate once and perform its action on that player. The square will not activate again on the same player until they have moved off of the square and re-moved onto it.

Some boards may have baddies - Bowser or Boo - roaming around on them. The starting location of the baddies for each board is specified in the board data file. The baddies will wander around the board squares and create trouble for Peach and Yoshi. As Peach and Yoshi travel around the board, they should avoid being in contact with baddies while the player avatars are in a *waiting to roll* state. By being in contact, we mean they are on the same square as a Boo or Bowser (i.e., share their exact x,y coordinates). Being in contact with a baddie while in the *waiting to roll* state may cause the player to have their stars or coins affected, depending on what the baddie decides to do. Bowser sometimes leaves droppings.

If Peach or Yoshi move onto an Event square, there is a 1 in 3 chance they will be given a Vortex projectile. Assuming Peach or Yoshi acquire a Vortex projectile during gameplay, the player may hit their appropriate firing key to shoot the projectile while in the *waiting to roll* state. A Vortex projectile will hit the first baddie it contacts (if any), and then the Vortex will disappear from the game. When hit, the baddie teleports to a random square on the game board. A Vortex will fly forward until either it hits a baddie or flies off the screen. Vortexes will pass over all objects other than baddies without affecting them.

[2] By "moves onto" a square, we mean that the player's avatar had the same x,y coordinates as the square while in the *walking* state.

Once gameplay begins, it is divided into small time periods called *ticks*. There are dozens of ticks per second (to provide smooth animation and game play).

During each tick of the game, your program must do the following:

- You must give each object – including Peach, Yoshi, Bowser, Boos, Vortex projectiles, squares, etc. - a chance to do something – e.g., move, shoot, activate, etc.
- You must delete/remove all dead/inactive objects from the game, e.g., projectiles that have either hit a baddie or flown off the screen.
- Your code may also need to introduce one or more new objects into the game – for instance, if the player instructs Yoshi to fire a Vortex projectile, his object might introduce a Vortex object into the game.
- You must update the game statistics line at the top of the screen, including the number of coins and stars that Peach and Yoshi each have, the remaining number of squares to move for the die rolls for each player, and whether Peach or Yoshi currently possesses a Vortex projectile to fire.
- You must check if the time has elapsed for the current game, and if so, end the current game and declare a winner.

The status line at the top of the screen must have the following format:

```
P1 Roll: 3 Stars: 2 $$: 15 VOR | Time: 75 | Bank: 9 | P2 Roll: 0 Stars: 1 $$: 22 VOR
```

There is no need to provide any leading zeros or spaces for numbers (e.g., "005" instead of "5"). The "VOR" should appear only if Peach or Yoshi have a Vortex projectile in their inventory. For example, if Peach (player P1) doesn't have a Vortex, the line would look like this:

```
P1 Roll: 3 Stars: 2 $$: 15 | Time: 75 | Bank: 9 | P2 Roll: 0 Stars: 1 $$: 22 VOR
```

Each of the stats of the status line must be separated from each other by exactly one space. For example, between "$$: 15" and "|" in the line above, there must be one space. You may find the *Stringstreams* writeup on the class web site to be helpful.

Your game implementation must play various sounds when certain events occur, using the *playSound()* method provided by our *GameWorld* class, e.g.:

```
// Make a sound effect when Peach shoots a Vortex
pointerToWorld->playSound(SOUND_PLAYER_FIRE);
```

You will find details on what sounds to play during what actions in the sections below. Constants for each specific sound, e.g., SOUND_PLAYER_FIRE, may be found in our *GameConstants.h* file.

Determining Object Overlap

In a video game, it's critically important to determine if two game objects come into contact with each other (are they close enough that they touch/overlap and therefore interact). For example, if Peach shoots a Vortex, how would the Vortex know if it has come into contact with a nearby Boo or Bowser as it flies across the screen? Similarly, squares that can activate (e.g., and give the players stars or coins) need to decide whether one of the players is standing on top of them. In Peach Party, there are two ways you'll need to determine object contact:

1. To detect if Peach or Yoshi have "landed on" or "moved onto" a square, or if Boo or Bowser have moved onto the same square as Peach or Yoshi, you determine if they overlap by comparing their x,y coordinates against each other. If the x,y coordinates of both game objects match exactly, then you have a contact. This is the easy case.
2. To detect if a Vortex projectile overlaps with a Boo or Bowser, you must check if any part of either game object's "bounding box" overlaps by even one pixel with the other. All game objects in Peach Party have a width/height of 16x16, so if an object is at location x=0, y=0, its "bounding box" would extend from 0,0 to 15, 15 inclusive. For instance, a game object at x=0, y=0 would be said to overlap with a game object at x=15,y=15 (whose bounding box goes from 15, 15 to 30,30 inclusive) because their corners overlap by at least one pixel. Be careful with detecting this kind of overlap - it's tricky!

So how does a video game work?

Fundamentally, a video game is composed of a bunch of game objects; in Peach Party, those objects include Peach, baddies (Bowser and Boos), squares of various types, and Vortex projectiles. 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 space, its own internal state (e.g., a baddie 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 Peach and Yoshi, the algorithm that controls these objects are the human players' own brains and hands, and the keyboard! In the case of other actors (e.g., a Bowser), 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 one simple 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 two pixels to the left), or change another objects' state (e.g., when a Bank square detects that it overlaps with Peach while she is in the *waiting to roll* state, it will grant Peach all of its coins). Typically, the behavior exhibited by an object during a

single tick is limited in scope to ensure that the gameplay is smooth and that things don't move too quickly and confuse the player. For example, a Bowser moves just a few pixels forward, rather than moving sixteen or more pixels per tick. A Bowser that moved, say, 20 pixels 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), the graphical framework that we provide animates the actors onto the screen in their new configuration. So if a Bowser changed its location from (x=16, y=50) to (x=14, y=50) (i.e., moved two pixels left), then our game framework would erase the graphic of the Bowser from location (16, 50) on the screen and draw the Bowser's graphic at (14, 50) instead. Since this process (asking actors to do something, then animating them to the screen) happens roughly 20 times per second, the user will see somewhat smooth animation.

Then, the next tick occurs, and each actor'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 Bowser doesn't move 3 inches away from where it was during the last tick, but instead moves a few millimeters 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 on the board.

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 need to be removed from the game. During a tick, new actors may be added to the game and actors who are tagged for removal must be removed from the game world and deleted.

Cleanup: Once the game (or level, in a multi-level game) is over, you must free all of the objects in the world (e.g., Peach, Bowsers, Boos, squares, Vortex projectiles, etc.).

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

```
prompt_the_user_for_which_board_to_play();// "press a # from 1-9"
initialize_the_game_world();        // you'll write this code

while (the play time has not elapsed)
{
  // 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 give you this code to handle the animation for you
        animate_each_actor_to_the_screen();
        sleep_for_50ms_to_give_the_user_time_to_react();
    }
    // The game has been completed - you're going to write this code
    cleanup_all_game_world_objects();   // you're going to write this
    declare_a_winner();                 // 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 board:
       if (the actor is still active/alive)
           tell the actor to do_something();
}
```

Your job in this project is to write the code that's shown in blue, as well as classes for all of the actors in the game (Peach, Yoshi, Bowser, Boo, squares, Vortexes, etc.).

Your game 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 monster from some typical video game might decide to do each time it gets asked to do something:

```
class Monster: public SomeOtherClass {
    public:
      virtual void doSomething() {
        If I see the player to the left:
          Switch the direction I'm facing to the left
          Move 4 pixels to the left
        If I see the player to the right:
          Switch the direction I'm facing to the right
          Move 4 pixels to the right
        ...
        If I've come into contact with the player
          Bite the player
        ...
      }
    ...
  };
```

And here's what the player's *doSomething( )* member function might look like in a typical video game:

```
class Player_Avatar: public …
{
public:
  virtual void doSomething() {
    if the player presses the left arrow key:
      set the avatar's direction to left
```

```
      move the avatar two pixels forward
    if the player presses the right arrow key:
      set the avatar's direction to right
      move the avatar two pixels forward
    ...
    if the player presses the space key:
      add a new projectile directly in front of the player
      decrement the ammunition count by one
  }
  …
};
```

What Do You Have to Do?

You must create a number of different classes to implement the Peach Party game. Your classes must work properly with our provided classes, and **you MUST NOT modify our provided 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* that is responsible for keeping track of your game world and all of the actors/objects (e.g., Peach, Bowser, Boos, squares and Vortex projectiles) that are inside the game.
2. You must create a class to represent players (Peach/Yoshi) in the game.
3. You must create classes for Bowser, Boos, squares (of all types), and Vortex projectiles, as well as any additional base classes (e.g., a Square base class if you find it convenient, since Squares share many things in common) that help you implement your actors.

You Have to Create the StudentWorld Class

Your *StudentWorld* class is responsible for orchestrating virtually all gameplay – it keeps track of the entire game world (each board and all of its inhabitants such as Bowser, Boos, squares of all types, Vortex projectiles, Peach, Yoshi, 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 Vortex flies off the screen or hits a baddie and disappears, etc.), and destroying *all* of the actors in the game world when the players finish playing the game.

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 for this project must *never* call any of these three functions (except that *StudentWorld*'s destructor may call *cleanUp()*). 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 (except possibly in the one place noted above).

Each time a game starts, 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 constructing a representation of the current board in your StudentWorld object and populating it with initial objects (e.g., squares, baddies, Yoshi and Peach), using one or more data structures that you come up with.

After the *init()* method finishes initializing your data structures/objects for the board, it **must** return `GWSTATUS_CONTINUE_GAME`.

Once the game's objects have been prepared 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., Peach, Yoshi, each Bowser, Boo, Square, Vortex, etc.) to try to do something: e.g., move themselves and/or perform their specified behavior. This method might also introduce new actors into the game, for instance adding a new Vortex projectile next to Peach if she fires. Finally, this method is responsible for disposing of (i.e., deleting) actors that need to disappear during a given tick (e.g., a Vortex that runs into a baddie and disappears). For example, if Bowser is shot by Peach's Vortex, then the Vortex object's state should be set to dead, and then after all of the alive actors in the game get a chance to do something during the tick, the *move()* method should remove that Vortex object 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 Peach and Yoshi complete the current game. The *cleanUp()* method is responsible for deleting all remaining actors (e.g., Peach and Yoshi, squares, baddies, Vortexes, etc.) that are currently managed by *StudentWorld* to prevent memory leaks. 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 Vortex that was added to the board by Yoshi when he fires) 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). You must **not** add any public data members.

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

```
void setGameStatText(string text);
int getAction(int playerNum);
void playSound(int soundID);
int getBoardNumber() const;
void setFinalScore(int stars, int coins);
void startCountDownTimer(int numSeconds);
int timeRemaining() const;
string assetPath() const;
```

The *setGameStatText()* method is used to specify what text is displayed at the top of the game screen, e.g.:

```
P1 Roll: 3 Stars: 2 $$: 15 | Time: 75 | Bank: 9 | P2 Roll: 0 Stars: 1 $$: 22 VOR
```

getAction() can be used to determine if the user has hit a key on the keyboard to control Peach or Yoshi's behaviors. This method checks whether the player indicated by the parameter (1 for Peach, 2 for Yoshi) hit a relevant key during the current tick. If a key that controls that player's action was hit (e.g., backslash to have Yoshi fire a Vortex), the function returns one of these constants (defined in *GameConstants.h*):

```
ACTION_LEFT
ACTION_RIGHT
ACTION_UP
ACTION_DOWN
ACTION_ROLL
ACTION_FIRE
```

If no key was pressed or a key that does not control that player's action, the function returns `ACTION_NONE`.

The *playSound()* method can be used to play a sound effect when an important event happens during the game (e.g., Peach shoots a Vortex or lands on a Bank Square). In *GameConstants.h* are constants (e.g., SOUND_PLAYER_FIRE) that describe what noise to make. The *playSound()* method is defined in our GameWorld class, which you will use as the base class for your StudentWorld class. Here's how this method might be used:

```
  // if Peach attacks a baddie
if (vortexOverlapsWithAbaddie())
    pointerToWorld->playSound(SOUND_HIT_BY_VORTEX);
```

getBoardNumber() can be used to determine which board number the users selected to play, and it returns a value from 1 to 9, since there are nine possible boards in our game. You can use this number to determine and load the proper board data file (e.g., board02.txt) for the board. Our framework collects the users' choice when the game starts and delivers it to you when you call this function.

setFinalScore() is used to report to the world how many stars and coins the winning player had at the end of the game just before the *move()* method returns a status of `GWSTATUS_PEACH_WON` or `GWSTATUS_YOSHI_WON`, so our framework can display the final result to the user. Here's how it could be used:

```
pointerToWorld->setFinalScore(winner_num_stars, winner_num_coins);
```

startCountDownTimer() starts a timer counting down from the number of seconds passed as its parameter (which should be 99, since the game is 99 seconds long). When testing, we may cause it to be called with a different value to lengthen or shorten the game.

timeRemaining() lets you find out how many seconds remain before the game ends.

Finally, *assetPath()* is used to get the path (aka directory) on your hard drive where the board data files, sprite graphics files, and sound files are located. You'll want to call this function in order to construct a path+filename to load your board data files before using the Board class.

init() Details

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

1. Initialize the data structures used to keep track of your game's world.
2. Allocate and insert Peach and Yoshi objects into the game world. Every time a game begins, Peach and Yoshi start out initialized in their starting location as specified by the current board data file.
3. Allocate and insert all of the other objects (e.g., squares, baddies, etc.) into the game world as described below.
4. Start the countdown timer for the 99-second game.

Your *init()* method must construct a representation of your world and store this in a *StudentWorld* object. Practically, this means that you'll load the board data file in using the Board class, and then determine where each "actor" (player, square and baddie) is supposed to be on the board. Then you'll create objects corresponding to each of these actors indicated in the board data file. It is **required** that you keep track of all of the actor objects in a **single** STL collection (e.g., a *list, set,* or *vector)*. To do so, we recommend using a container of pointers to the actors. If you like, your *StudentWorld* object may keep separate pointers to the Peach and Yoshi objects rather than keeping pointers to them in the container with the other actor pointers; Peach and Yoshi are the **only** actor pointers allowed to not be stored in the single actor container. The *init()* method may also initialize any other *StudentWorld* member variables it needs (such as the amount of money in the bank).

The *init()* method must use our provided Board class (found in Board.h) to load the specified board data file into a Board object, and use this to populate the current board with objects at the proper locations. The following types of objects must be populated:

- Peach
- Yoshi
- Baddies like Boo and Bowser
- Squares

The Board class provides the initial locations of all game objects on the board. Since each board is a grid that's 16 squares wide by 16 squares high, all coordinates used by the Board class are between 0 and 15. However, the screen where you actually display your graphics is 256 pixels x 256 pixels. So when creating game objects that will be displayed on the screen, you'll need to convert the grid values from the Board class into pixel values. In practice, this just means multiplying each grid location by 16. So if the Board class says there's a Bowser at location (x1, y1) then you'd actually create your Bowser object at location (SPRITE_WIDTH * x1, SPRITE_HEIGHT * y1).

The *init()* method returns `GWSTATUS_BOARD_ERROR` if the board data file doesn't exist or if the file is improperly formatted. Otherwise, *init()* returns `GWSTATUS_CONTINUE_GAME`. These constants are defined in *GameConstants.h*.

You must not call the *init()* method yourself. Instead, our framework code will call this method when it's time for a new game to start.

move() Details

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

1. Check if the game is over (i.e., the time remaining is no longer positive). If so:
    a. Play the SOUND_GAME_FINISHED sound using *playSound*().
    b. Call *setFinalScore()* function to report the number of stars and coins of the winning player.
    c. If Peach won, return `GWSTATUS_PEACH_WON`. If Yoshi won, return `GWSTATUS_YOSHI_WON`. If the game was a tie game, pick a winner randomly and return one of the above return values.
2. It must ask all of the actors that are currently active in the game world to do something (e.g., ask each Bowser to move itself, ask each Vortex projectile to check if it overlaps with a baddie, and if so, teleport the baddie to a random square, give Peach and Yoshi both a chance to move, roll, or shoot a Vortex, etc.).
3. It must then delete any actors that have become inactive/dead during this tick (e.g., a Vortex that flies off the screen and needs to be removed from the game).
4. It must update the status text on the top of the screen with the latest information (e.g., each player's stats).
5. If the game is not over, then the function must return `GWSTATUS_CONTINUE_GAME`.

The return value, `GWSTATUS_CONTINUE_GAME`, indicates that the game is not yet over. Therefore, the game play should continue normally for the time being. In this case, the framework will display the newly-moved sprites on the screen, advance to the next tick, and call your *move()* method again.

The final return values, `GWSTATUS_PEACH_WON` and `GWSTATUS_YOSHI_WON` indicate that the game is over and that we have a winner. If your *move()* method returns one of these values, then the current game is over, and our framework will call your *cleanUp()* method to destroy the contents of the board. Our framework will then display the results of the game and exit.

IMPORTANT NOTE: The skeleton code that we provide to you is hard-coded to return a `GWSTATUS_NOT_IMPLEMENTED` status value from our dummy version of the *move()* method. Unless you implement something that returns `GWSTATUS_CONTINUE_GAME` your game will not display any objects on the screen! So if the screen just immediately tells you that *init* returns a bad status once you try to start playing, you'll know why!

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

```
int StudentWorld::move()
{
    // The term "actors" refers to all actors, e.g., Peach, Yoshi,
    // baddies, squares, vortexes, etc.

    // Give each actor a chance to do something, incl. Peach and Yoshi
    for each of the actors in the game world
    {
        if (actor[i] is still active/alive)
        {
            // tell that actor to do something
          actor[i]->doSomething();
        }
    }

    // Remove newly-inactive actors after each tick
    remove inactive/dead game objects

    // Update the Game Status Line
    update display text   // update the coins/stars stats text at screen top

    if (time has expired for the game)
    {
      play end of game sound;
      if (yoshi won)
      {
        setFinalScore(yoshi_stars, yoshi_coins);
        return GWSTATUS_YOSHI_WON;
      }
      else  // peach won
      {
        setFinalScore(peach_stars, peach_coins);
        return GWSTATUS_PEACH_WON;
      }
    }

    // the game isn't over yet so continue playing
    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, fire, etc.). Actors include Peach, Yoshi, baddies like Bowser and Boo, Vortexes, and squares.

Your *move()* method must iterate over every actor that's active in the game (i.e., held by your *StudentWorld* object) and ask it to do something by calling a member function in the actor's object named something like *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 Bowser might move two pixels left
- Yoshi might shoot a Vortex projectile
- Peach might roll the die
- A previously-fired Vortex may disappear due to smacking into a baddie or flying off the screen

It is possible that an actor (e.g., a Vortex) may need to "die" or become inactive during a tick. If so, other live actors processed during the tick must not interact with the actor after it has died (e.g., a Vortex that just hit a Bowser and which has thus become inactive must not interact with a subsequent Boo or Bowser).

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 active, remove them from your container of active actors, and use a C++ delete expression to free their objects (so you don't have a memory leak). So if, for example, if a Vortex impacts a Bowser, then the Vortex should have its state changed to inactive, and at the end of the tick, its *pointer* should be removed from the StudentWorld's container of active objects, and the Vortex object should be deleted (using a C++ delete expression) to free up memory for future actors that will be introduced later in the game. (Hint: Each of your actors could maintain an active/inactive status member variable.)

Detecting When the Game Is Over

Your *move()* method must determine when no more time remains in the game (determined by calling *timeRemaining()*). Hint: In your *StudentWorld::init()* method you can call *startCountDownTimer()*. Then, each time your *StudentWorld::move()* method runs, you can call *timeRemaining()* and check how many seconds are left.

cleanUp() Details

When your *cleanUp()* method is called by our game framework, it means that the game is over. Every actor in the entire game (including Peach and Yoshi) must be deleted and removed from the *StudentWorld's* container of active objects, resulting in an empty board.

You must not call the *cleanUp()* method yourself when the game ends. Instead, this method will be called by our code automagically when *move()* returns an appropriate status indicating the game is over.

Board Data File

As mentioned, every board of Peach Party may have a different layout. The layout for each board is stored in a text data file that you can edit with Windows notepad, vi, emacs, or Mac's textedit. The file "board01.txt" holds the details for the first board, "board02.txt" holds the details for the second board, etc. These board data files are stored in the Assets directory along with all bitmaps and sound files.

An example data file is shown below - you can modify our data files to create *dank* new boards, or add your own new board data files to add new boards, if you like.

board02.txt:

```
B+++++    +++++b
+    ^++++b    +
-    +    +    -
+    $    +    +
+    +    +    +
+    +    +    !
+    +    +    +
-    +    +    +
!    +    +    +
*    +    !    +
+    +    +    +
+    !    +    -
+    b++++v    +
+    +    -    +
@+!+++    ++++!+
```

As you can see, the data file contains a 16x16 grid of different characters that represent the different actors/things on the board. Valid characters for your board data file are:

The + character represents a blue coin square that gives 3 coins to a player that lands upon it.

The - character represents a red coin square which takes away 3 coins from a player that lands upon it.

The * character represents a star square which grants the player a star and takes away 20 coins when it's landed upon.

The **!** character represents an event square. Event squares result in the player that landed upon them randomly either (a) being teleported, (b) having their position swapped with the other player, or (c) getting a Vortex projectile that they can fire later in the game.

The **$** character represents a bank square. Players that just pass over a bank square contribute to the bank, and players that land on the bank square get all the money that was contributed previously.

The **@** character specifies three game objects at the same location:

1. The starting location of Peach
2. The starting location of Yoshi
3. A blue +3 coin square (underneath Peach and Yoshi)

The **B** character represents two game objects at the same location:

1. Bowser's starting square
2. A blue +3 coin square (underneath the Bowser)
   Note: There must be at most one Bowser on a given board.

The **b** character represents two game objects at the same location:

1. A Boo's starting square
2. A blue +3 coin square (underneath the Boo)
   Note: There may be more than one Boo on a given board.

The **>** character represents a directional square that forces a player that passes over/lands on the square to change its movement direction to 0 degrees (toward the right of the screen) and continue moving in this direction. This should set the direction the sprite is facing to 0 degrees.

The **<** character represents a directional square that forces a player that passes over/lands on the square to change its movement direction to 180 degrees (toward the left of the screen) and continue moving in this direction. This should set the direction the sprite is facing to 180 degrees.

The ^ character represents a directional square that forces a player that passes over/lands on the square to change its movement direction to 90 degrees (toward the top of the screen) and continue moving in this direction. This should set the direction the sprite is facing to 0 degrees, even though its direction of movement will be 90 degrees.

The **v** (lowercase V) character represents a directional square that forces a player that passes over/lands on the square to change its movement direction to 270 degrees (toward the bottom of the screen) and continue moving in this direction. This should set the direction the sprite is facing to 0 degrees, even though its direction of movement will be 270 degrees.

All **space** characters represent empty locations where Peach, Yoshi, Bowsers and Boos must NOT move on the board.

You must NOT have tab characters in your board data files, so make sure your text editor inserts only spaces, not tabs.

You may assume that all boards have no dead-ends where Yoshi and Peach can get stuck; so all boards must essentially have loops of squares.

You may assume that any time there is a directional square pointing in a particular direction, there will always be a valid square next to the directional square in the specified direction. A directional square will never point to an empty space with no square.

You may assume that there will be exactly one **@** character on the board, and that Peach and Yoshi always start out a board with a valid square to their right, since they start out with a rightward direction of movement.

The Board Class

We are providing you with a class that can load board data files for you. The class is called *Board* and may be found in our provided *Board.h* file. Here's an example of some of the class's functionality:

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

void StudentWorld::someFunc()
{
  Board bd;

  string board_file = assetPath() + "board01.txt";
  Board::LoadResult result = bd.loadBoard(board_file);
  if (result == Board::load_fail_file_not_found)
    cerr << "Could not find board01.txt data file\n";
  else if (result == Board::load_fail_bad_format)
    cerr << "Your board was improperly formatted\n";
  else if (result == Board::load_success) {
    cerr << "Successfully loaded board\n";

    Board::GridEntry ge = bd.getContentsOf(5, 10);  // x=5, y=10
    switch (ge) {
     case Board::empty:
       cout << "Location 5,10 is empty\n";
       break;
     case Board::boo:
       cout << "Location 5,10 has a Boo and a blue coin square\n";
       break;
     case Board::bowser:
```

```
        cout << "Location 5,10 has a Bowser and a blue coin square\n";
        break;
      case Board::player:
        cout << "Location 5,10 has Peach & Yoshi and a blue coin square\n";
        break;
      case Board::red_coin_square:
        cout << "Location 5,10 has a red coin square\n";
        break;
      // etc…
    }
  }
}
```

Hint: You will want to use our *Board* class to load the board specification in your *StudentWorld* class's *init()* method. The *assetPath()* and *getBoardNumbers()* methods your *StudentWorld* class inherits from *GameWorld* might also be useful too!

You Have to Create the Classes for All Actors

Peach has a number of different actors, including:

- Peach
- Yoshi
- Coin Squares
- Star Squares
- Directional Squares
- Bank Squares
- Event Squares
- Dropping Squares
- Boo
- Bowser
- Vortexes

Each of these actor types can occupy your various boards and interact with other game actors within the visible screen view.

Now of course, many of your game actors will share things in common – for instance, every one of the actors in the game (Bowsers, Boos, Peach and Yoshi) are allowed to move only along the squares. Certain objects like squares, Vortexes and baddies (Boo and Bowser) "activate" when they come into contact with a Player, etc.

It is therefore your job to determine the commonalities between your different actor classes 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 avoiding duplicate non-trivial code or a data member** – if you find yourself writing the same (or largely similar) code or duplicating member variables 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 Coin Square section and the Star Square section, or in the Boo and Bowser sections) you must make sure to identify common behaviors and move these into proper base classes. NEVER duplicate non-trivial behaviors (aka methods and member variables) across classes that can be moved into a base class! A non-trivial behavior is one that is more than a single statement long. Try to place common behaviors to the nearest common base class (e.g., if all baddies spit fire, then put the spit_fire() method in a Baddie base class rather than a more general Actor 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 Bowser: public Actor  // or some subclass of Actor
{
public:
	…
};

class CoinSquare: public Actor  // or some subclass of Actor
{
public:
	…
};
```

GraphObject is a class that we have defined that helps hide the ugly logic required to graphically display your actors on the screen. If you don't derive 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,
            int startDirection = right, int depth = 0, double size = 1.0);
int getX() const;           // in pixels (0-255)
int getY() const;           // in pixels (0-255)
void moveTo(int x, int y);  // in pixels (0-255)
int getDirection() const;        // sprite angle in degrees (0-359)
void setDirection(Direction d);  // sprite angle in degrees (0-359)
void getPositionInThisDirection(int angle, int distance,
                                int& newX, int& newY) const;
void moveAtAngle(int angle, int distance);
```

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 **unless** they are marked as virtual in our base class.

```
GraphObject(int imageID,
            int startX,          // column first: x
            int startY,          // then row: y
            int startDirection   // 0, 90, 180, 270 / right, up, left, down
            int depth = 0,       // graphical depth
            double size = 1.0)   // don't worry about size for this project
```

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 Boo, Bowser, squares, Yoshi, Peach, 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 (these constants are defined in our provided header file *GameConstants.h*). Notice that you pass the coordinates as x, y (i.e., column, row starting from bottom left, and ***not*** row, column). You may also specify the initial direction an object is facing as an angle between 0-359 degrees (though you'll only use values of 0, 90, 180, and 270 for this project). Objects with lower graphical depths will be drawn on top of those of higher depths.

For the *imageID*, you must pass in one of the following IDs (found in *GameConstants.h*):

IID_PEACH
IID_YOSHI
IID_BLUE_COIN_SQUARE
IID_RED_COIN_SQUARE
IID_DIR_SQUARE
IID_EVENT_SQUARE
IID_BANK_SQUARE
IID_STAR_SQUARE
IID_DROPPING_SQUARE
IID_BOWSER
IID_BOO
IID_VORTEX

If you derive your game objects from our *GraphObject* class, they will be displayed on screen automatically by our framework (e.g., a Boo image will be drawn to the screen at the *GraphObject's* specified x,y coordinates if the object's Image ID is IID_BOO).

The classes you write MUST NOT have a data member that is an image ID value or any value somehow related/derived from the image ID value in any way or you will get a Zero on this project. That includes strings, ints, enums, or any other clever data elements you can think of. Only our GraphObject class may store the image ID or related value.

You MUST not use the image ID to identify object types, e.g., determine that a particular actor is a Bowser by checking if its image ID is IID_Bowser). Nor may you store other similar data (e.g., a string "Bowser", enum, etc.) based on the image ID to identify your object types. For hints on how to distinguish between different actors, see the Object-Oriented Programming Tips section later in this document!

DO NOT TRY TO FIND A CREATIVE SOLUTION THAT GETS AROUND THE ABOVE REQUIREMENTS. OUR ANSWER WILL BE "YOU CAN'T DO THAT. DO IT USING PROPER OOP STYLE."

getX() and *getY()* are used to determine a *GraphObject's* current location on the board. Since each *GraphObject* maintains its own (x, y) location, this means that your derived classes **must NOT** also have x or y member variables, but instead use these functions and *moveTo()* from the *GraphObject* base class.

moveTo() is used to update the location of a *GraphObject* on the board and also updates our sprite frames if the image has multiple animated frames. For example, if a Bowser's movement logic dictates that it should move one pixel to the left, you could do the following:

```
moveTo(getX()-1, getY());  // move one pixel to the left
```

getDirection() is used to determine the direction a *GraphObject* is facing, and returns a value of *0-359*.

setDirection() is used to change the direction a *GraphObject* is facing and takes a value between 0 and 359. For example, you could use this method and *getDirection()* to adjust the direction of an actor when it decides to move in a new direction. Note that an actor can be facing in one direction, but move in a totally different direction using *moveTo()*. The angle of movement and the angle the actor is facing are allowed to be different.

getPositionInThisDirection() is used to calculate the position that is a particular distance at a particular angle from the object's current position. As an example, if an object at (128, 80) wants to compute the position that is 2 pixels in the upward direction (90 degrees), saying

```
int xnew, ynew;
getPositionInThisDirection(up, 2, xnew, ynew);
```

sets xnew to 128 and ynew to 82. (*GraphObject* defines the constants *right*, *up*, *left*, and *down*, as 0, 90, 180, and 270, respectively.) This function does not itself move the object.

moveAtAngle() is used to update the location of a *GraphObject* by moving it the specified number of pixels in the specified direction. For example, if a Vortex's movement logic dictates that it should move left 4 pixels, you could say the following:

```
moveAtAngle(left, 4);  // move the object 4 pixels left
```

You ***must*** use the *moveTo()* or *moveAtAngle()* methods 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).**

The Player Avatar (for Peach and Yoshi)

Here are the requirements you must meet when implementing the Player Avatar class.

What Player Avatar Object Must Do When It Is Created

(Note - only those items highlighted in blue are required for part #1 of the project; all items are required for part #2 of the project)

When it is first created:

1. A Player Avatar object must have an image ID of IID_PEACH or IID_YOSHI.
2. A Player Avatar object has a sprite direction of 0 degrees.
3. A Player Avatar object has a starting walk direction of right.
4. A Player Avatar object has a starting (x,y) position based on the board data file. Your *StudentWorld* object can pass in that position when constructing this object.
5. A Player Avatar must keep track of whether it is player 1 (uses left side of keyboard) or player 2 (uses right side of keyboard).
6. A Player Avatar object has a graphical depth of 0.
7. A Player Avatar object has a size of 1.
8. A Player Avatar object starts with zero coins and zero stars.
9. A Player Avatar object starts with no Vortex projectile.
10. A Player Avatar has a ticks_to_move value of 0.
11. A Player Avatar starts out in the *waiting to roll* state.

What Player Avatar Must Do During a Tick

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

1. If the Avatar is in the *waiting to roll* state:
    a. If the Avatar has an invalid direction (due to being teleported):
        i. Pick a random valid direction (there must be a square in that direction).
        ii. Update the direction the Avatar's sprite faces based on the walk direction: if the walk direction is left, the sprite direction must be 180 degrees; for all other walk directions, the sprite direction must be 0 degrees.
        iii. Continue with step b.

b. See if the user pressed a key using *getAction()*.
c. If the action is ACTION_ROLL:
    i. die_roll = random number from 1 to 10, inclusive
    ii. ticks_to_move = die_roll * 8
    iii. Change the Avatar's state to the *walking* state.
    iv. Continue with step 2 below.
d. If the action is ACTION_FIRE:
    i. Introduce a new Vortex projectile on the square directly in front of the Avatar in the Avatar's current walk direction.
    ii. Play the SOUND_PLAYER_FIRE sound using *GameWorld*'s *playSound()* method.
    iii. Update your Avatar so it no longer has a Vortex to shoot.
e. Else if action is anything else (because the user didn't press a key or pressed any other key):
    i. Return immediately.

2. If the Avatar is in the *walking* state:
    a. If the Avatar is directly on top of a directional square[3]:
        i. Update the Avatar's walk direction to the direction specified by the directional square.[4]
        ii. Update the direction the Avatar's sprite faces based on the walk direction: if the walk direction is left, the sprite direction must be 180 degrees; for all other walk directions, the sprite direction must be 0 degrees.
        iii. Continue with step d below.
    b. Else if the Avatar is directly on top of a square at a fork (with multiple directions where it could move next):
        i. See if the user pressed a key using *getAction()*.
        ii. If the user pressed a key and chose a valid direction to move (i.e., the direction they chose has a square on it and is not the direction they came from):
            1. Update the Avatar's walk direction to the user-provided direction.
            2. Update the direction the Avatar's sprite faces based on the walk direction: if the walk direction is left, the sprite direction must be 180 degrees; for all other walk directions, the sprite direction must be 0 degrees.
            3. Continue with step d below.
        iii. Else if the user didn't press a key or didn't select a valid direction:
            1. Immediately return.
    c. Else if the Avatar can't continue moving forward in its current direction:
        i. Update the Avatar's walk direction so it can turn to face a new direction perpendicular to the current walking direction.

---

[3] Meaning the X,Y coordinates are identical.

[4] Alternatively, you could have the Directional Square object inform the player object to change its direction once the player lands on top of the Directional Square, rather than the player object checking what square it's on top of and changing its direction itself.

1. Always prefer up over down if both options are available.
2. Always prefer right over left if both options are available.

ii. Update the direction the Avatar's sprite faces based on the walk direction: if the walk direction is left, the sprite direction must be 180 degrees; for all other walk directions, the sprite direction must be 0 degrees.

iii. Continue with step d.

d. Move two pixels in the walk direction.

e. Decrement the ticks_to_move count by 1.

f. If ticks_to_move is 0 then:

i. Change the Avatar's state to the *waiting to roll* state.

What Player Avatar Must Do In Other Circumstances

The following actions can be performed on Peach and Yoshi by other actors in the game (e.g., a Coin Square might increase the Avatar's coins by 3). Each of the items below might be implemented by a separate method inside the Avatar class (or perhaps one of its superclasses) that can be called by other classes in your game.

- The Avatar can have coins added to or removed from its inventory.
- The Avatar can have stars added to or removed from its inventory.
- The Avatar can have its coins swapped with the other player's Avatar.
- The Avatar can have its stars swapped with the other player's Avatar.
- The Avatar can have a Vortex power given to it (only one at a time).
- The Avatar can have its position and movement state swapped with the other player's Avatar:
    - This swaps the current ticks_to_move, the roll/walk state, the walk direction, and the sprite direction with the other player's Avatar (so they switch positions on the board, and continue moving the same direction/distance (or waiting to roll) as the other player was about to do).
- The Avatar can be teleported to another square:
    - Pick at random a square on the board. (Note that the Avatar must not be teleported to a location without a square.)
    - Update the Avatar's location to that square.
    - Make the Avatar's "walk direction" invalid, enabling the Avatar to pick a direction once it lands rather than forcing the Avatar to use its previous direction. You may leave the Avatar's sprite direction the same.
- Avatars are not impactable (e.g., by Vortex projectiles).

Getting Input From the User

Since Peach Party is a *real-time* game, you can't use the typical *getline* or *cin* approach to get a user's key press within your Player Avatar'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 *getAction()* that we provide in our *GameWorld* class (from which your *StudentWorld* class is derived) to get input from the player[5]. This function rapidly checks to see if the user has hit a key relevant to the player. If so, the function returns an int (e.g., `ACTION_UP`, `ACTION_ROLL`, etc.) indicating the action corresponding to that key. Otherwise, the function returns `ACTION_NONE`, meaning that no relevant key was hit. This function could be used as follows:

```
void Avatar::doSomething()
{
    ...
    switch (getWorld()->getAction(m_playerNumber))
    {
      case ACTION_NONE:
        ... no key relevant to this player was hit ...
        break;
      case ACTION_LEFT:
        ... change player's direction to left ...
        break;
      case ACTION_FIRE:
         ... add Vortex object in front of player...;
        break;
      ...
    }
    ...
}
```

Coin Square

Coin Squares are squares on the game board that, when landed on by a player, either give coins to, or deduct coins from, that player.

What a Coin Square Must Do When It Is Created

When it is first created:

1. A Coin Square object must have an image ID of IID_BLUE_COIN_SQUARE (for Coin Squares that give coins to the player) or IID_RED_COIN_SQUARE (for Coin Squares that deduct coins to the player).
2. A Coin Square object has a starting (x, y) position based on the board data file.

[5] Hint: Since your Avatar class will need to access the *getAction()* 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 Avatar'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 *getAction()* method.

3. A Coin Square object may be configured to either grant 3 coins, or deduct three coins from each player that lands on it.
4. A Coin Square object has a sprite direction of 0 degrees.
5. A Coin Square object has a graphical depth of 1.
6. A Coin Square object has a size of 1.
7. Hint: You may want your Coin Square, or one of its base classes, to have an "alive" flag that indicates whether it's still capable of being activated (since Bowser can convert a Coin Square into a Dropping Square, necessitating destruction of the Coin Square and removal from the *StudentWorld*'s actor container).

What a Coin Square Must Do During a Tick

A Coin Square must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the square must:

1. Check if it is still alive/active (since Coin Squares can be destroyed by Bowsers). If it is not active, the Coin Square must do nothing and immediately return.
2. Check if a new player has landed upon the square (but not simply passed over it).
    a. A "new" player is one that just landed on the square and has not yet activated the square. If the player remains on the square for more than one tick, they are no longer considered a "new" player, unless they first leave the square and then return to it at a later time.
3. If a new player has landed on the square and it's a space that gives coins, then:
    a. Give that player 3 coins.
    b. Play the SOUND_GIVE_COIN sound using *playSound()*.
4. If a new player has landed upon it and it's a space that takes coins, then:
    a. Deduct 3 coins from the player (or as many as the player has, if fewer than 3; a player can never have negative coins).
    b. Play the SOUND_TAKE_COIN sound using *playSound()*.

What a Coin Square Must Do In Other Circumstances

- Coin Squares are not impactable (e.g., by Vortex projectiles).

Star Square

Star Squares are squares on the game board that, when landed on or moved onto by a player, deduct 20 coins from the player and then give them one star. Star Squares only activate if the player has at least 20 coins and the player *lands on* or *moves onto* them.

What a Star Square Must Do When It Is Created

When it is first created:

1. A Star Square object must have an image ID of IID_STAR_SQUARE.
2. A Star Square object has a starting (x, y) position based on the board data file.
3. A Star Square object has a sprite direction of 0 degrees.
4. A Star Square object has a graphical depth of 1.
5. A Star Square object has a size of 1.

What a Star Square Must Do During a Tick

A Star Square must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the square must:

1. Check if a new player has landed upon the square (finished the roll) OR has moved onto the square (before finishing the roll).
    a. A "new" player is one that just landed on or moved onto the square and has not yet activated the square. If the player remains on the square for more than one tick, they are no longer considered a "new" player, unless they first leave the square and then return to it at a later time.
2. If a new player has landed on or moved onto the square then:
    a. If the player has fewer than 20 coins:
        i. Immediately return.
    b. Else:
        i. Deduct 20 coins from the player.
        ii. Give 1 star to the player.
        iii. Play the SOUND_GIVE_STAR sound using *playSound()*.

What a Star Square Must Do In Other Circumstances

- Star Squares are not impactable (e.g., by Vortex projectiles).

Directional Square

Directional Squares are squares on the game board that indicate what direction a player must move as it passes over or lands upon the Directional Square. For example, if a Directional Square is found at a fork on the board, it indicates a mandatory direction to follow.

What a Directional Square Must Do When It Is Created

When it is first created:

1. A Directional Square object must have an image ID of IID_DIR_SQUARE.
2. A Directional Square object has a starting (x, y) position based on the board data file.

3. A Directional Square object has a sprite direction (0, 90, 180, 270) which is specified by the board data file.
4. A Directional Square object has a graphical depth of 1.
5. A Directional Square object has a size of 1.
6. A Directional Square object has a "forcing direction": Which direction does it force the player to change their direction to? This would be the same as the sprite direction.

What a Directional Square Must Do During a Tick

A Directional Square must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the square must:

1. Check if a player has landed upon the square (finished the roll) OR has moved onto the square (before finishing the roll).
2. If a player has landed on or moved onto the square then:
    a. Inform the player that their walking direction must be set to the Directional Square's forcing direction.

Alternatively, you could have the Player Avatar object check if it lands on or moves upon a Directional Square object and change its direction itself, rather than have the Directional Square handle this interaction.

What a Directional Square Must Do In Other Circumstances

- Directional Squares are not impactable (e.g., by Vortex projectiles).

Bank Square

Bank Squares are squares on the game board that do the following:

- Any time a new player passes over (but does not land on) the Bank Square, the Bank Square deducts up to 5 coins from the player and deposits it in a central bank account that is shared by all of the Bank Squares.
- Any time a new player lands on (but does not pass over) the Bank Square, the Bank Square gives the player ALL of the coins from the central bank account, and resets the bank account balance to zero.

If there are multiple Bank Squares on the board, they all deposit into a central bank account. (Hint: You can store this in your *StudentWorld* object on behalf of all the Bank Squares; each Bank Square doesn't need to have its own account.)

What a Bank Square Must Do When It Is Created

When it is first created:

1. A Bank Square object must have an image ID of IID_BANK_SQUARE.
2. A Bank Square object has a starting (x, y) position based on the board data file.
3. A Bank Square object has a sprite direction of 0 degrees.
4. A Bank Square object has a graphical depth of 1.
5. A Bank Square object has a size of 1.

What a Bank Square Must Do During a Tick

A Bank Square must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the square must either:

1. Check if a new player has *landed upon* the square (but not simply passed over it).
    a. A "new" player is one that just landed on the square and has not yet activated the square. If the player remains on the square for more than one tick, they are no longer considered a "new" player, unless they first leave the square and then return to it at a later time.
2. If a new player has landed upon the square then:
    a. Get the balance of the bank and give that many coins to the player.
    b. Set the balance of the bank to zero.
    c. Play the SOUND_WITHDRAW_BANK sound using *playSound()*.
3. Check if a player has moved onto the square (but not landed on it).
4. If a player has moved onto the square then:
    a. Deduct 5 coins from the player (or as many as the player has, if fewer than 5; a player can never have negative coins).
    b. Add to the central bank account as many coins as were deducted from the player.
    c. Play the SOUND_DEPOSIT_BANK sound using *playSound()*.

What a Bank Square Must Do In Other Circumstances

- Bank Squares are not impactable (e.g., by Vortex projectiles).

Event Square

Event Squares are squares on the game board that perform a random action on Peach or Yoshi when they land upon the square, including:

1. Teleporting the player who landed on the square to another square on the board.
2. Swapping the positions and movement states of the player who landed on the square with the other player.
3. Giving the player who landed on the square a Vortex projectile with which they can later shoot a baddie.

What an Event Square Must Do When It Is Created

When it is first created:

1. An Event Square object must have an image ID of IID_EVENT_SQUARE.
2. An Event Square object has a starting (x, y) position based on the board data file.
3. An Event Square object has a sprite direction of 0 degrees.
4. An Event Square object has a graphical depth of 1.
5. An Event Square object has a size of 1.

What an Event Square Must Do During a Tick

An Event Square must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the square must:

1. Check if a player has landed upon the square (finished the roll and ended on the Event Square)
2. If a player has landed upon the square, then the square chooses randomly amongst the following actions and perform one:
    a. Option #1: Inform the player that they have been teleported to another random square on the board (the player will handle the details of teleporting itself properly). Play the SOUND_PLAYER_TELEPORT sound using *playSound()*.
    b. Option #2: Instruct the player who landed on the square to swap its position and movement state with the other player. The following state (or your equivalent) must be swapped between the players:
        i. x, y coordinates
        ii. the number of ticks left that the player has to move before completing their roll
        iii. the player's walk direction
        iv. the player's sprite direction
        v. the player's roll/walk state

        NOTE: After the players have been swapped, the Event Square must NOT activate on the other player who just had its position swapped onto the Event Square.

        Play the SOUND_PLAYER_TELEPORT sound using *playSound()*.
    c. Option #3: Give the player a Vortex projectile (if they don't already have one; a player can have at most one Vortex projectile at a time). Play the SOUND_GIVE_VORTEX sound using *playSound()*.

What a Event Square Must Do In Other Circumstances

    d. Event Squares are not impactable (e.g., by Vortex projectiles).

Dropping Square

Dropping Squares are squares on the game board that, when ~~stepped in~~ landed on by a player, randomly deduct either 10 coins from the player or 1 star from the player.

What a Dropping Square Must Do When It Is Created

When it is first created:

1. A Dropping Square object must have an image ID of IID_DROPPING_SQUARE.
2. A Dropping Square object must have its (x, y) location specified for it by Bowser when Bowser deposits the new square. Your Bowser object can pass in that (x, y) position when constructing this object.
3. A Dropping Square object has a sprite direction of 0 degrees.
4. A Dropping Square object has a graphical depth of 1.
5. A Dropping Square object has a size of 1.

What a Dropping Square Must Do During a Tick

A Dropping Square must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the square must:

1. Check if a new player has *landed upon* the square (but not simply passed over it)
    a. A "new" player is one that just landed on the square and has not yet activated the square. If the player remains on the square for more than one tick, they are no longer considered a "new" player, unless they first leave the square and then return to it at a later time.
2. If a new player has landed upon the square:
    a. Pick one of the following actions and execute it:
        i. Option #1: Deduct 10 coins from the player (or as many as the player has, if fewer than 10; a player can never have negative coins).
        ii. Option #2: Deduct one star from the player if the player has at least one star.
    b. Play the SOUND_DROPPING_SQUARE_ACTIVATE sound using *playSound()*.

What a Dropping Square Must Do In Other Circumstances

- Dropping Squares are not impactable (e.g., by Vortex projectiles).

Bowser

Bowsers are baddies who roam around the game board and cause trouble for Peach and Yoshi

What a Bowser Must Do When It Is Created

When it is first created:

1. A Bowser object must have an image ID of IID_BOWSER.
2. A Bowser object has a starting (x, y) position based on the board data file.
3. A Bowser object has a sprite direction of 0 degrees.
4. A Bowser object has a starting walking direction of right.
5. A Bowser object has a graphical depth of 0.
6. A Bowser object has a size of 1.
7. A Bowser object starts with an initial travel distance of 0 pixels.
8. A Bowser object has two states, *Walking* and *Paused*, and starts out in the *Paused* state.
9. A Bowser has a pause counter which tells him how long to wait on a square when he has finished walking. The pause counter starts out at a value of 180.

What a Bowser Must Do During a Tick

A Bowser must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the square must:

1. If the Bowser is in the *Paused* state, then:
    a. If Bowser and a Player Avatar are on the same square, and the Avatar is in a *waiting to roll* state, then:
        i. There is a 50% chance that:
            1. Bowser will cause the Avatar to lose all their coins and stars and will play the SOUND_BOWSER_ACTIVATE sound using *playSound()*.

        Note: Bowser will only activate once per Player Avatar per location until one or both actors move off the square; this prevents repeated Bowser actions on the same player.
    b. Decrement the pause counter.
    c. If the pause counter reaches zero, then:
        i. squares_to_move = random number from 1 to 10, inclusive
        ii. ticks_to_move = squares_to_move * 8
        iii. Pick a new random direction for Bowser to walk that is legal (there is a square in that direction that Bowser may move onto).
        iv. Update the direction the Bowser's sprite faces based on the walk direction: if the walk direction is left, the sprite direction must be

180 degrees; for all other walk directions, the sprite direction must be 0 degrees.
v. Set Bowser to the *Walking* state.
vi. Continue with step 2.

2. If Bowser is in the *Walking* state:
    a. If Bowser is directly on top of a square AND he's at a fork where there are multiple directions he could choose to move, then:
        i. Pick a new random direction for Bowser to walk that is legal (a legal direction is one where there is a square in that direction that Bowser may move onto).
        ii. Update the direction the Bowser's sprite faces based on the walk direction: if the walk direction is left, the sprite direction must be 180 degrees; for all other walk directions, the sprite direction must be 0 degrees.
    b. Else if Bowser is directly on top of a square and there is not a square in front of Bowser to move onto next, then he's at a turning point:
        i. Update the Bowser's walk direction so he can turn to face a new direction perpendicular to the current walking direction.
            1. Always prefer up over down if both options are available.
            2. Always prefer right over left if both options are available.
        ii. Update the direction the Bowser's sprite faces based on the walk direction: if the walk direction is left, the sprite direction must be 180 degrees; for all other walk directions, the sprite direction must be 0 degrees.
    c. Move two pixels in the walk direction.
    d. Decrement ticks_to_move by one.
    e. If ticks_to_move is zero, then:
        i. Set Bowser's state to the *Paused* state.
        ii. Set the pause counter to 180 (ticks).
        iii. There is a 25% chance that:
            1. Bowser will deposit a dropping by asking the *StudentWorld* object to remove the square underneath him and insert a new Dropping Square in its place and play the SOUND_DROPPING_SQUARE_CREATED sound using *playSound()*.

<u>What a Bowser Must Do In Other Circumstances</u>

- A Bowser object is impactable. When impacted:
    - The Bowser object must randomly teleport itself to another square on the board.
    - The teleported Bowser object must have a walking direction of right and sprite direction of 0 degrees.
    - The Bowser object will transition immediately into the *Paused* state.
    - The Bowser object will set its pause ticks to 180.

Boo

Boos are baddies who roam around the game board and cause troubles for Peach and Yoshi

What a Boo Must Do When It Is Created

When it is first created:

1. A Boo object must have an image ID of IID_BOO.
2. A Boo object has a starting (x, y) position based on the board data file.
3. A Boo object has a sprite direction of 0 degrees.
4. A Boo object has a starting walking direction of right.
5. A Boo object has a graphical depth of 0.
6. A Boo object has a size of 1.
7. A Boo object starts with an initial travel distance of 0 pixels.
8. A Boo object has two states, *Walking* and *Paused*, and starts out in the *Paused* state.
9. A Boo has a pause counter which tells him how long to wait on a square when he has finished walking. The pause counter starts out at a value of 180.

What a Boo Must Do During a Tick

A Boo must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the square must:

1. If the Boo is in the *Paused* state, then:
    a. If Boo and a Player Avatar are on the same square, and the Avatar is in a *waiting to roll* state, then:
        i. Boo will randomly execute one of the following two options:
            1. Swap that player's coins with the other player.
            2. Swap that player's stars with the other player.
        ii. Play the SOUND_BOO_ACTIVATE sound using *playSound()*.

    Note: Boo will only activate once per Player Avatar per location until one or both actors move off the square; this prevents repeated Boo actions on the same player.
    b. Decrement the pause counter.
    c. If the pause counter reaches zero, then:
        i. squares_to_move = random number from 1 to 3, inclusive
        ii. ticks_to_move = squares_to_move * 8
        iii. Pick a new random direction for Boo to walk that is legal (there is a square in that direction that Boo may move onto).
        iv. Update the direction the Boo's sprite faces based on the walk direction: if the walk direction is left, the sprite direction must be

180 degrees; for all other walk directions, the sprite direction must be 0 degrees.
v. Set Boo to the *Walking* state.
vi. Continue with step 2.

2. Else If Boo is in the *Walking* state:
    a. If Boo is directly on top of a square AND he's at a fork where there are multiple directions he could choose to move, then:
        i. Pick a new random direction for Boo to walk that is legal (a legal direction is one where there is a square in that direction that Boo may move onto).
        ii. Update the direction the Boo's sprite faces based on the walk direction: if the walk direction is left, the sprite direction must be 180 degrees; for all other walk directions, the sprite direction must be 0 degrees.
    b. Else if Boo is directly on top of a square and there is not a square in front of Boo to move onto next, then he's at a turning point:
        i. Update the Boo's walk direction so he can turn to face a new direction perpendicular to the current walking direction.
            1. Always prefer up over down if both options are available.
            2. Always prefer right over left if both options are available.
        ii. Update the direction the Boo's sprite faces based on the walk direction: if the walk direction is left, the sprite direction must be 180 degrees; for all other walk directions, the sprite direction must be 0 degrees.
    c. Move two pixels in the walk direction.
    d. Decrement ticks_to_move by one.
    e. If ticks_to_move is zero, then:
        i. Set Boo's state to the *Paused* state.
        ii. Set the pause counter to 180 (ticks).

<u>What a Boo Must Do In Other Circumstances</u>

- A Boo object is impactable. When impacted:
    - The Boo object must randomly teleport itself to another square on the board.
    - The teleported Boo object must have a walking direction of right and sprite direction of 0 degrees.
    - The Boo object will transition immediately into the *Paused* state.
    - The Boo object will set its pause ticks to 180.

Vortex

A Vortex is a projectile that can be shot by Peach and Yoshi. When a Vortex impacts a baddie (Bowser or Boo) it causes them to teleport to another random square on the board, and the Vortex disappears from the game.

What a Vortex Must Do When It Is Created

When it is first created:

1. A Vortex object must have an image ID of IID_VORTEX
2. A Vortex object must have its (x, y) location specified for it by Peach or Yoshi when they create the new Vortex.
3. A Vortex object has a direction that is specified by the firing actor.
4. A Vortex object has a sprite direction of 0 degrees.
5. A Vortex object has a graphical depth of 0.
6. A Vortex object has a size of 1.
7. A Vortex object starts out in the active state.

What a Vortex Must Do During a Tick

A Vortex must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the Vortex object must do the following:

1. The Vortex object must check if it is active, and if not, immediately return.
2. The Vortex object must move two pixels in its current movement direction (which was specified when the Vortex was created).
3. If the Vortex object leaves the boundaries of the screen (e.g., its X coordinate is less than 0 or is greater than or equal to VIEW_WIDTH, or its Y coordinate is less than 0 or is greater than or equal to VIEW_HEIGHT) then it must become non-active (and thus destroyed by StudentWorld at the end of the current tick).
4. The Vortex object must see if it currently *overlaps*[6] with an object that can be impacted by a Vortex object. Right now, only Boo and Bowser objects may be impacted by a Vortex (but you might imagine other actors we might want to add in the future could also be "impacted", so design your classes for extensibility). If the Vortex overlaps with one or more impactable objects, then:
    a. If there are more than one object that the Vortex overlaps with (e.g., two Boos on the same square), pick one of them to be hit (it's up to you how to pick).
    b. The Vortex object will inform the other object that it has been impacted. The impacted object can then decide what to do once it's impacted (each object can figure out how to react to being impacted: See the Boo and Bowser sections; right now, they teleport themselves).
    c. The Vortex object will immediately set its own state to inactive/dead.
    d. The Vortex object will play the SOUND_HIT_BY_VORTEX sound using *playSound()*.

[6] By overlap, we mean that the sprites overlap by even one or more pixels. Remember that each sprite is 16 pixels wide by 16 pixels high. So for example, a Vortex at location X=16, Y=18 would overlap with a Boo at X=16, Y=32, or a Bowser at X=30, Y=16

What a Vortex Must Do In Other Circumstances

- A Vortex object is not impactable by other Vortex objects.

Object Oriented Programming Tips

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 NEVER use the image ID (e.g., IID_BOWSER, IID_BLUE_COIN_SQUARE, IID_VORTEX, etc.) to determine the type of an object or store the imageID inside any of your objects as a member variable. Doing so will result in a score of ZERO for this project. You must also not use any equivalent, e.g., adding strings like "IID_BOO", enums, ints or morse code to your classes. Don't try to be creative and get around this.**

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

Don't do this:

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

Do this instead:

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

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

Don't do this:

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

Do this instead:

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

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

Don't do this:

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

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

Do this instead:

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

```
};
```

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

6. **Never make a method public if it is 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 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 doSomething()
    {
      doCommonThingA();
      passStinkyGas();
      pickNose();
      doCommonThingB();
    }
};

class ShinyRobot: public Robot
{      …
  public:
    virtual void doSomething()
    {
      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 and sound 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 macOS). 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:/CS32Project3/Assets"` or `"/Users/fred/CS32Project3/Assets"`).

To build the game, follow these steps:

For Windows

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

If you build and run your program from within Xcode, the Assets directory should be in the directory yourProjectDir*/DerivedData/*yourProjectName*/Build/Products/Debug* (e.g., */Users/ethel/CS32Project3/DerivedData/PeachParty/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/ethel*).

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 Peach Party 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., a base class that accommodates Peach and Yoshi, both types of baddies, squares, projectiles, etc.):
    i. It must have a constructor that initializes the object appropriately.
    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 Coin Square class, derived in some way from the base class described in 1 above:
    i. For part #1, you are responsible only for those items highlighted in blue in the Coin Square section.
    ii. You may add any public/private member functions and private data members to your Coin Square 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 Player Avatar class, derived in some way from the base class described in 1 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 Avatar – see Player Avatar section for more details on how to initialize the Avatar. For part #1, you are only responsible for those items highlighted in blue.
    ii. It must have a limited version of a *doSomething()* method that implements the functionality highlighted in blue.
    iii. You may add other public/private member functions and private data members to your Avatar class as you see fit, so long as you use good object-oriented programming style (e.g., you must not duplicate functionality across classes). You need not implement any other functionality of your Avatar class.
4. A limited version of the *StudentWorld* class.
    i. Add any private data members to this class required to keep track of all game objects/actors. Right now all of those game objects will just be Coin Squares and a single Avatar (Peach) but eventually you'll need to include all of your actors.
    ii. Implement a constructor for this class that initializes your data members.
    iii. Implement a destructor for this class that frees all dynamically allocated objects, if any, that have not yet been freed at the time the *StudentWorld* object is destroyed.
    iv. Implement the *init()* method in this class. It must load up the board data file using our provided Board class, and then create just Coin Square objects and an Avatar object for Peach at the right locations, and insert them into your StudentWorld data structures. Make sure to place a Coin Square at the same x,y coordinates as Peach, since Peach and Yoshi always start on a Coin Square. The positions of each of the squares and Peach must be set based on the contents of the board data file. Your *init()* method may ignore any other objects like Yoshi, Boos, Bowsers, other types of squares, etc.; it need only deal with Peach and Coin Squares for part #1.
    v. Implement the *move()* method in your *StudentWorld* class. During each tick, it must ask Peach and other actors (just Coin Squares for now) to do something. Your *move()* method does not have to deal with any actors other than Peach and the Coin Squares.
    vi. Implement a *cleanUp()* method that frees any dynamically allocated data that was allocated during calls to the *init()* method or the *move()* method (i.e., it should delete all your allocated Coin Squares and Peach). 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 CS 131 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, PlayerAvatar, Player, Avatar, or something similar would be reasonable names for what we called the Player 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 '1' to begin playing, your executable displays the board with Peach in her proper starting position, and Coin Squares in their proper positions as specified in the board data file. If your classes work properly, you should be able to hit the die-roll key (Tab) and each time Peach will move between 1-10 squares around the board. (The *board01.txt* board is a simple loop of blue coins.)

Your Part #1 solution may actually do more than what is specified above; for example, if you are making good progress, try to make your Coin Square class activate so it gives Peach 3 coins when she lands upon the square. Just make sure that what you have builds and has at least as much functionality as what's described above, and you may turn that in instead.

Note, the Part #1 specification above doesn't require you to implement any Boos, Bowsers, other types of squares or Vortex projectiles (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 Peach, Coin Square, and StudentWorld classes still work properly and that your program still builds and meets at least 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 will turn in a zip file containing nothing more than these four files:

```
Actor.h             // contains base, Player, and Coin Square 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 Peach Party 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. We may also devise a simple test framework that runs under g32; if we do, your code **must build without errors** in that framework. If it does not also run without errors, that indicates some fundamental problem that will probably cost you a lot of points. 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, report.doc, 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"` in *main.cpp*.)

You must turn in a report that contains the following:

1. A description of the control flow for the interaction of a player avatar and a bank square. 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 Vortex class.” or “My Bowser doesn’t work correctly yet so it behaves like a Boo right now.”
3. A list of 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 57 or so in GameController.cpp with a larger value, like 20 or 50:

```
static const int MS_PER_FRAME = 5;
```

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 and doesn't crash right away! If we can sort of play your game, but it’s not complete or perfect, that’s better than if it won’t even build!

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

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

GOOD LUCK!