Project #3
NachMan

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

Time due:

Part 1: Thursday, February 24 at 9 pm
Part 2: Tuesday, March 1 at 9 pm

Table of Contents

Introduction

NachenGames corporate spies have learned that SmallSoft is planning to release a new Pac-Man game, called NachMan, and would like you to program an exact copy so NachenGames can beat SmallSoft to the market. To help you, the NachenGames spies have managed to steal a prototype NachMan executable file and several source files from the SmallSoft headquarters, so you can see exactly how your version of the game should work (see attached 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.)

NachMan is a simplified version of the original Pac-Man game where the player controls the NachMan character, who navigates through a set of mazes, eating pellets and dodging monsters. The goal of each level is for NachMan to eat all of the regular pellets and power pellets while avoiding getting eaten (and hence killed) by one of the four roving monsters. While the monsters are normally deadly to NachMan, if he eats a power pellet while roving through the maze, then all of the monsters in a normal state become vulnerable for a limited period of time. During this time, NachMan may eat the monsters for extra points. Once eaten, a monster returns to starting position in the maze, where he immediately reverts to his normal (and dangerous) state and again begins hunting NachMan. A specified number of ticks after NachMan eats a power pellet, all *vulnerable* monsters revert into their normal state and are again deadly to NachMan. The game play continues until either NachMan has lost all three of his lives or the player has solved every level of the game.

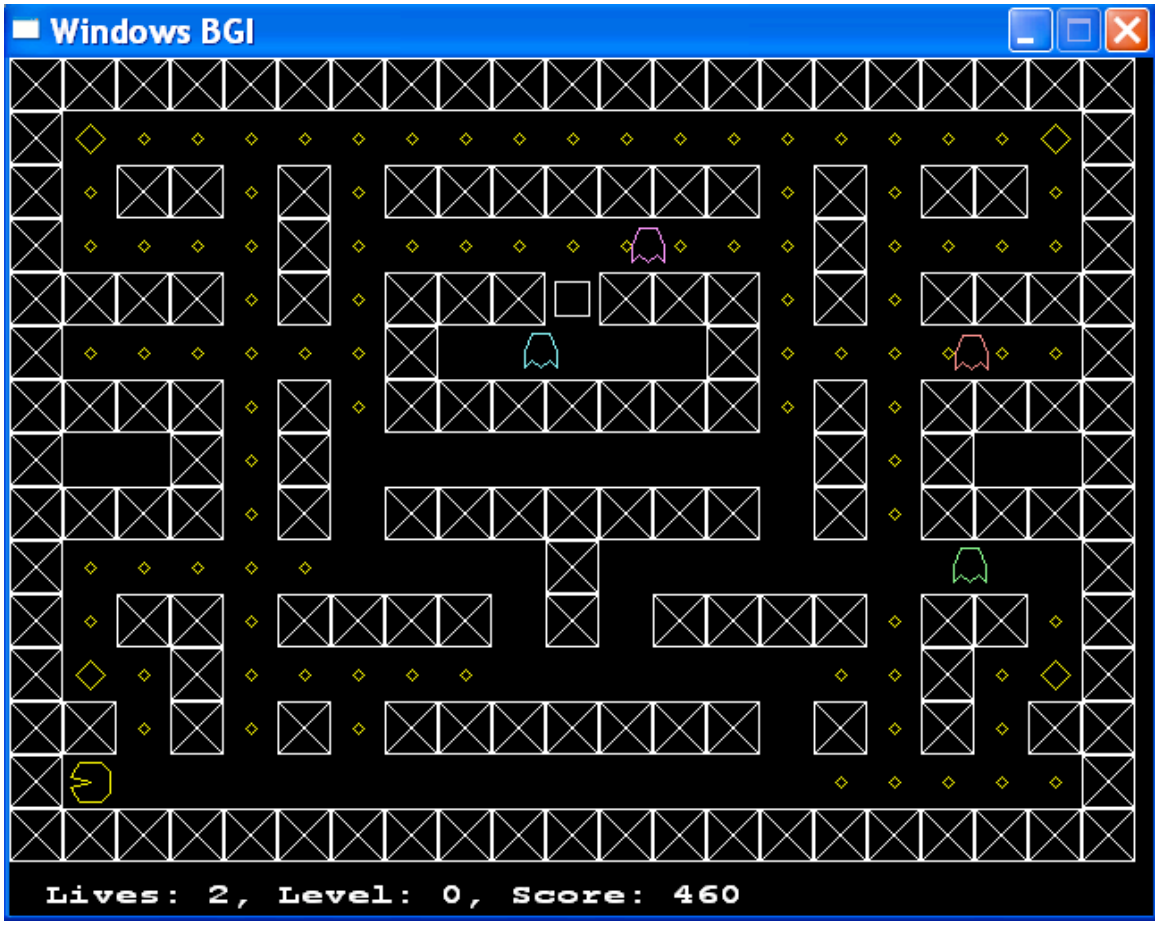
Upon starting a new game, NachMan is placed in his starting position in the first maze of the game and prompted to begin. From here, NachMan must work his way through the maze, eat the pellets and avoid the monsters. Once NachMan finishes eating all of the pellets on a level, the player is congratulated, prompted to hit enter to continue, and then advanced to play the next level. A player who finishes the last level of the game wins, and the game ends.

On screen, NachMan is represented by a yellow circle with a big mouth (like that of the founder of NachenGames). On each level, NachMan will encounter four different types of monsters that he will have to try to avoid: Inky, Stinky, Dinky, and Clyde.
When NachMan eats a power pellet, all of the monsters become vulnerable for a short period of time. If a monster is currently vulnerable, then it is displayed in a light blue color on the screen and may be eaten by NachMan. Otherwise, if the monster is in its normal deadly state, it is displayed in its default color:

1. Inky (shown on-screen as a light red monster)
2. Stinky (shown on-screen as a light green monster)
3. Dinky (shown on-screen as a light magenta monster)
4. Clyde (shown on-screen as a light cyan monster)

Finally, if a monster is currently returning to its starting position after being eaten, then it should be displayed using a light gray color. Each of these monsters has different behaviors and reacts differently to NachMan. You will find more details about them below in the Monsters section of this document.

Here is an example of what the game display might look like:



Here is what each of the different display icons are:

- Each box with an X in it is a wall. **Neither NachMan nor monsters can move through a wall.**
- Each maze has a cage door which only monsters can go through. In the above maze, the cage door is the small white rectangle just above and to the right of Clyde (the light-cyan monster). **NachMan may never move onto or through a cage door. Monsters may move onto or through the cage door.**
- Each small yellow diamond represents a normal food pellet.
- A large yellow diamond represents a power pellet. When NachMan eats a power pellet, the monsters change into a vulnerable state for a predefined number of game ticks.
- A blank square is used to represent an empty spot in the maze.

To move NachMan, the player uses the keyboard's arrow keys.

Game Details

NachMan starts out a new game with 3 lives and can continue to play until he either exhausts his lives or he completes all of the game's mazes.

The game consists of one or more mazes, with each maze's layout specified by a simple text file (named maze0.txt, maze1.txt, maze2.txt, etc.). Feel free to edit our maze data files in Notepad to get an idea of how they're laid out. You may also add your own mazes if you like. The game will proceed from maze0 to mazeN until all maze files have been exhausted; then the game is over.

Each maze data file contains information such as where all of the walls are, where the pellets are, where the monster cage is, and the position in the maze where the monsters and NachMan should all start at the beginning of each level (the "home squares" for the monsters and NachMan). All mazes are 21 squares wide by 15 squares high.

When the player starts a new maze, NachMan is placed in the maze at its designated home position with no default direction (i.e. NachMan should not move until the player hits a directional key). Similarly, when starting a new maze, all monsters must be placed in their starting home positions. Both NachMan's and the monsters' designated home positions are specified inside in each maze data file. When a new level begins, all monsters start in their normal state, and are therefore dangerous to NachMan.

Once a new level has been prepared – NachMan and all of the monsters are in their proper positions and in the proper states – the game-play begins. At this point, both NachMan (controlled by the human player) and all of the monsters (controlled by algorithms that you, the programmer, provide) are allowed to move around the maze. NachMan continues to eat pellets until either he clears the entire level of pellets or is caught by a monster (i.e., he lands on the same square as a monster in a *normal* state).

Game-play is divided into a set of "ticks." During each tick, NachMan is allowed to move one square in any direction (potentially eating a pellet, running into a monster, etc), and each monster is allowed to move one square in any direction (each monster's movement algorithm decides which adjacent square to move to). During each tick, as NachMan and monsters move, there are several possible things that can happen:

1. NachMan may move into an empty square.
2. NachMan may eat a regular pellet and earn 10 points.
3. NachMan may eat a power pellet and earn 100 points, placing the monsters in a vulnerable state (see details below).
4. NachMan may run into the same square as a monster and either eat the monster or be killed by the monster (depending on whether the monster is *vulnerable* or not).

After a tick, any movements of NachMan and the monsters are animated onto the computer screen for the player to see, and then the next tick begins. (Our classes take care of all the animation for you.)

If NachMan succeeds in clearing all of the pellets from the maze, then the current maze is discarded, the new maze is loaded, and the player proceeds with the next level, until all levels have been played and the game is over.

If NachMan is killed by a monster, his number of remaining lives is decremented by 1. (He started with 3.) If he still has at least one life left, then NachMan is returned to his home position in the current maze, all of the monsters are returned to their home positions in the maze, and both NachMan and all of the monsters have their states reset (NachMan will no longer be facing a particular direction, all of the monsters will have their state set to normal and their colors returned to their default color, and NachMan will have his state reset from dead to not-dead). Then the player is prompted to continue and given another chance to clear the remaining pellets from current maze. On the other hand, if NachMan is killed by a monster and has no lives left, then the game is over.

During game-play, the player controls the direction of NachMan by pushing a directional keys (either the arrow keys or keys on the numeric keypad: '8' for up, '2' for down, '4' for left, and '6' for right). Once given a direction to move, NachMan should continue to move in that direction until either the player changes NachMan's direction by pressing another directional key or NachMan runs into a wall or a monster cage door (or dies due to landing on the same square as a *normal* monster). If NachMan runs into either a wall or a cage door, he will stop moving until the player selects a valid new direction by pressing a directional key.

Each of the monsters moves according to a different algorithm, and each monster's algorithm further depends on what state the monster is in.

Monsters have four different states they can be in:

- *normal*: When a monster is in the *normal* state, it will kill NachMan if the monster and NachMan both land on the same square. Each monster has a specific movement algorithm (specified later in this spec) that it must use when it's in the *normal* state. All monsters start each level in the *normal* state. When NachMan dies and restarts the current level, all monsters must start in the *normal* state as well. Normal monsters are displayed in their default color (e.g., light cyan for Clyde).
- *vulnerable*: Each monster has a different, more evasive movement algorithm (specified later in this spec) that it must use when it's in the *vulnerable* state – in other words, it shouldn't be chasing NachMan because it's afraid of being eaten. Why? When a monster is in the *vulnerable* state, it will be eaten by NachMan if they both land on the same square. Monsters become *vulnerable* when NachMan eats a power pellet, and they stay *vulnerable* for a predefined number, G, of game ticks or until they're eaten by NachMan. If a monster has not been eaten before the expiration of G game ticks, it returns to the *normal* state. If, however, a

monster is eaten by NachMan while in a *vulnerable* state, it will transition during the same tick to the *monsterdie* state. Vulnerable monsters are displayed in a light blue color.

- *monsterdie:* Once a vulnerable monster has been eaten, it transitions into this state. A monster stays in this state for just one tick. It then immediately transitions directly into the *returntohome* state.
- *returntohome:* When a monster is in the *returntohome* state, its only goal is to move back to its home square (where it started in the maze at the beginning of the level) as quickly as possible. It will not change state (e.g., to *vulnerable* or *normal*), even if NachMan eats another power pellet, until it reaches its home position in the maze. Once the monster reaches its home position, it then transitions back to the *normal* state. Monsters in the *returntohome* state are displayed in a gray color.

Each time NachMan eats a regular pellet in the maze (i.e. walks onto the same square as a pellet), he receives 10 points.

When NachMan eats a power pellet, he receives 100 points and all monsters that are not in the *returntohome* or *monsterdie* states transition during the same tick into the *vulnerable* state for a specified number, G, of game ticks (see details in the NachMan section below for how to compute G and what a "tick" is).

If NachMan ***runs into a vulnerable monster*** *or* ***a vulnerable monster runs into NachMan***, then the player receives 1000 points and the monster changes state to the *monsterdie* state. In this state, the monster must make a dying noise, and not move for one tick. Then the monster immediately transitions into the *returntohome* state. In this state, during each subsequent tick, it must move one square toward its home position in the cage. Once the monster reaches its home position, the monster returns to the *normal* state and can continue hunting NachMan using its normal behavior.

If NachMan eats a power pellet while a monster is returning to its home position or is in the dying state (i.e., in the *returntohome* or *monsterdie* states), then that monster is not switched into a vulnerable state (i.e., the monster simply continues to return home). A monster that was just eaten and in the *returntohome* state will not revert back to the *normal* state until it reaches its home (even if its number G of vulnerability ticks reaches zero). See the Monster section below for more details on these requirements.

If ***NachMan runs into a monster*** *or* ***a monster runs into NachMan*** while the monster is in a *normal* state, then NachMan dies and loses one life. If NachMan moves onto a square with a normal pellet *and* a normal monster, then it is your choice whether NachMan will eat the pellet first and then die, or NachMan will die first and not consume the pellet. If NachMan dies and has additional lives, then the player is prompted to hit enter (by the code we provide) and continue playing (with all monsters starting out at the designated monster home square in a *normal* state and NachMan starting at the designated NachMan home square). If NachMan has exhausted all of his lives, then the game is over.

Two or more monsters may occupy the same space in the maze at the same time without any problem (however, given the limitations in our graphical display class, you may only see one of the monsters in the cell; the background monsters may be hidden beneath the foreground monster). Therefore, when the player directs NachMan onto a square, it is possible that multiple monsters will be present on that same square. If NachMan moves onto a square with multiple *vulnerable* monsters, the player should be given points for all of the *vulnerable* monsters eaten on the square and each of the *vulnerable* monsters on the square should transition first to the *monsterdie* state and then, a tick later, to the *returntohome* state so it can move back to the starting point in its cage. If NachMan moves onto a square with both normal and vulnerable monsters, then it is your choice whether to have NachMan die first without eating the vulnerable monsters, or give the player points for eating the vulnerable monsters, and then have NachMan die.

NachMan may move onto the following types of squares: empty squares, squares with regular pellets, squares with power pellets. NachMan must not move onto or through wall squares or cage door squares.

Monsters may move onto any square that NachMan can move onto. In addition, they may move onto cage door squares. They still must not move through walls.

Each monster uses a different algorithm to decide how to move through the maze and chase NachMan. For example, when in a normal state, Clyde moves randomly through the maze and doesn't intentionally chase NachMan (even if he sees NachMan only a few squares away). Clyde will, however, run away from NachMan when he is *vulnerable*. In contrast, Dinky will move randomly through the maze until he sees NachMan in his line of sight. If Dinky sees NachMan and is in a *normal* state, then he will move toward NachMan to try to kill him. A full listing of all monster behaviors is included in the sections below.

So how does a video game work?

Fundamentally, a video game is comprised of a bunch of objects, like NachMan, the monsters, etc. Each object has its own x,y location, its own internal state (whether the monster is vulnerable or not, how many lives NachMan has left, etc.), and its own special algorithm to control its actions in the game. In the case of NachMan, the algorithm that controls the NachMan object is the player's own brain!

Once the game begins, the passage of time is divided into "ticks." A tick is a unit of time, for example, $1/20^{th}$ of a second. During a given tick, every actor (e.g., NachMan, monsters) in the game has an opportunity to do something (e.g., move to an adjacent square in the maze, eat a power pellet, die, switch from a normal to a vulnerable state, etc.). During each tick, the player is prompted to hit a key to control the NachMan object (e.g., changing the direction of NachMan). After the current tick is over and all objects have had a chance to adjust their state, they are then displayed on the screen in their latest configuration. Then, the next tick occurs, and each object is again allowed to do something, etc.

Assuming the ticks are quick enough (a small fraction of a second), and the actions performed by the objects are subtle enough (i.e., a NachMan doesn't move 3 inches away from where it was during the last tick, but instead moves 1 millimeter away), when you display each of the objects on the screen after each tick, it looks like each object is performing a continuous series of fluid motions.

This is what the simplified ***main game loop*** of a game usually looks like:

```
vector<Actor*> actors;
...
while (gameIsNotOver())
{
     for (int i = 0; i < actors.size(); i++)
         actors[i]->DoSomething();

     updateDisplay(actors);// plot all actors on the screen

     Sleep(50);                // sleep 50ms so the game isn't too fast
                               // (this gives us 20 ticks per second)
}
```

As you can see, in the example above, each object has a *DoSomething* function. In this function, each actor (e.g., a monster) can decide what to do. For example, here is some pseudocode showing what a monster might decide to do each time it is asked to move:

```
class Monster: public …
{
public:
      void DoSomething()
      {
            If I'm in a normal (aggressive) state
                  if (NachMan is to my left)
                        then adjust my x location left by one
                  else if (NachMan is to my right)
                        then adjust my x location right by one
                  ...
            Else if I'm in a vulnerable state
                  If (Nachman is to my left)
                        then adjust my x location right by one
                  ...
      }

      ...
};
```

And here's what NachMan's class might look like:

```
class NachMan: public …
{
public:
      void DoSomething()
      {
            Try to get player input (if any is available)
```

```
                if (the player pressed the UP key AND
                    there is no wall in the up direction)
                      then adjust my y location up one square
                ...

                if (NachMan landed on square with a power pellet)
                      remove the power pellet from the maze
                      for each monster
                            adjust monster's status to vulnerable
                ...
                if (NachMan and normal monster are at the same place)
                      then set NachMan's status to DEAD
          }
    ...
    };
```

What Are We Providing?

We've provided you with a skeleton program composed of a set of classes, described below. ***You must NOT modify these classes in any way – you may only make the changes/additions we explicitly describe!***

Maze

This class is responsible for keeping track of the current maze, including where all of the walls are, which pellets have been eaten and which are still available, the x,y locations of the home positions of NachMan and all of the Monsters, etc. It class also keeps track of all the actors in the game (e.g., the Monsters and NachMan). This class is also responsible for displaying the maze and all of the actors onto the screen in a graphical format – your classes DO NOT have to worry about displaying objects on the screen. We do all the hard graphics work so you can focus on perfecting your inheritance and polymorphism skills.

You must NOT modify any code in the Maze class.

You'll be interested in (and probably need to use) the following public member functions that are provided by our Maze class:

`GridContents GetGridContents(int x, int y)`: This function returns the contents of a particular x,y location in the maze. See the definition of the *GridContents* enumerated type in Maze.h for possible values (e.g., EMPTY, WALL, PELLET, etc.).
`void SetGridContents(int x, int y, GridContents nGC)`: This function sets the contents of a particular x,y location in the maze. For instance, you could specify that location 6,9 is EMPTY once NachMan eats the pellet in that square.
`int GetRemainingFood()`: Returns the total number of pellets and power pellets that still must be eaten by NachMan for the current level to be completed.

`int GetMonsterStartX(), int GetMonsterStartY()`: Each maze data file specifies where all monsters should start out in the maze – their home square. This function returns the specified home X,Y location for all of the monsters in the current maze (they all have the same home square).
`int GetNachManStartX(), int GetNachManStartY()`: Each maze data file specifies where NachMan should start out in the maze – his home square. This function returns the specified starting X,Y location for NachMan in the current maze.
`bool GetNextCoordinate(int nCurX, int nCurY, int &nNextX, int &nNextY)`: This function is used by monsters to help them figure out how to return to their home square after being eaten by NachMan. Each time one of your monsters calls this function, it passes in the monster's current X,Y coordinates. The function then determines what direction your monster should move in to efficiently return to its home square. The function returns *false* if the monster has arrived on its home square (indicating that no more movement is needed). The function returns *true* if the monster has not yet reached its home square and still needs to move (the function then provides you with the new X,Y coordiantes where your monster should move next). You are going to implement this function's behavior yourself in a subclass of Maze called MyMaze (details to follow).

World

This class is in charge of running the actual game and keeping track of all of the game's actors and its current maze.

A World object starts by loading a new maze from a data file, adding all of the actors into the maze, and then running the main game loop (***which you're going to write – it's going to look something like the one shown in the section above***). As the main game loop runs, NachMan and all of the monsters run around through the maze until either NachMan dies or NachMan eats all of the pellets on the screen.

Once a level is complete (due to NachMan dying or because the player completed the current level), then World object frees the current level, loads up the next level (if any), and runs the game loop again.

You must NOT modify any code in the World class.

Your actors will need to access the other actors and the contents of the maze to work properly.

For example, your NachMan object might need to know if it can move onto square 11, 5 in the maze – is there a wall there, a power pellet there, etc? Or your NachMan might need to know where all of the Monsters are so it can check to see if it ran into any of them and it should die. Or your Monsters might need to determine the current location of NachMan to see if they've run into him so they can kill him.

To do all of these things, your NachMan and Monster classes can request this information from the World class. You'll be interested in (and need to use) the following public member functions that are provided by our World class:

`NachMan* getNachMan()`**:** Returns a pointer to the World's NachMan object.
`Monster* getMonster(int monsterNum)`**:** This can be used to obtain a pointer to the i^{th} monster object, where the monsters are numbered from 0 to 3. For example, Inky might be monster number 0, Stinky might be monster number 1, etc.
`Maze* getMaze()`:This function can be called to obtain a pointer to the maze object that represents the game's current maze. Your actor classes will need to access the contents of the maze in order to determine where they can move, and to remove pellets from the maze as they're eaten by NachMan.
`int GetLevel()`**:** Returns the current level number in the game, where the first level is level 0 (not level 1).

SoundFX

The SoundFX class can be used to play sound clips to enhance the gameplay. You'll want to use this class to add certain sound effects (which we'll specify) to the game.

You'll need to use the `SoundFX::playNachManSound()` function to provide sound effects during the game. This function accepts a single parameter which is the ID# of the sound effect to play. Valid sound effect IDs are:

`PAC_SOUND_START:` `The game is starting.`
`PAC_SOUND_SMALL_EAT:` `NachMan ate a regular pellet.`
`PAC_SOUND_BIG_EAT:` `NachMan ate a monster or power pellet.`
`PAC_SOUND_DIE:` `NachMan died by running into a monster.`

The playNachManSound is a static member function of the SoundFX class. You can look at our code to see how to properly call this function in your code.

You must NOT modify any code in the SoundFX class.

Misc. Other Classes

We also have built a number of other classes, such as GraphManager (which handles all display of graphics to the screen), and GraphObject (which is used to draw the actual maze and actors on the screen), but you don't need to directly use these classes. You may look at them if you'd like to see how the graphical elements of the game works, but this is not required to do well on the project.

You must NOT modify any code in these other provided classes.

What do YOU have to do?

You have to (simply?) fill in the blanks to complete the NachMan program:

1. You must create a set of classes to represent the game's actors (e.g., NachMan and the monsters). These classes ***must*** work properly with our provided World and Maze classes. These classes ***must*** behave as described in the sections below. You must ***not*** modify our classes in order to make your classes work. Your classes must work with our classes as they are.
2. You must create a subclass of our Maze class, called MyMaze, and add a member function called *GetNextCoordinate( )* to it (the details of this function will be described later).
3. You must create a subclass of our World class, called MyWorld, and add a function called *RunLevel( )* to it (the details of this function will be described later).

Detailed Requirements

This section describes requirements of the NachMan game. These are all mandatory project requirements.

Note: All data members in your classes must be private; none may be public or protected.

Actors

You will need to define a set of classes to represent your NachMan and the monsters and place these classes in actor.h and actor.cpp. What you will notice is that all of our classes rely upon classes called *Actor*, *NachMan*, *Monster*, *Inky*, *Stinky*, *Dinky* and *Clyde*, which represent the game's actors. Since all Actors have certain basic attributes/functions in common, you're going to have to define an *Actor* base class and add this basic functionality to this class. You'll then derive your NachMan and Monster classes from this base class. Of course, there are some behaviors that are common to all monsters, and some behaviors that are specific to each monster, so you'll want to put common functions/data into your base Monster class and then derive and customize your individual monster classes (Inky, Clyde, etc…) from Monster. The same holds true for NachMan vs. the Monsters. You may derive intermediate superclasses, as required, so long as your classes work with our program (without having to modify our provided code).

When defining your Actor class hierarchy, you should use proper class design techniques, including: using const where appropriate, using virtual and pure virtual functions as appropriate, using public and private properly, etc. If two classes (e.g., Inky and Dinky) share similar behaviors, then you should move these behaviors into an appropriate

superclass rather than duplicating their code in both subclasses. You will be docked points, for example, if you define a function with the same exact behavior across multiple subclasses, since this behavior belongs in a superclass.

You may define any set of public and private member functions for your Actor, NachMan, and Monster classes as long as you use proper Object Oriented design principles ***and*** *your classes work with our provided classes*.

The Actor Class

The Actor class can be used to represent all of your game's actors (e.g., NachMan and the Monsters). You may implement it any way you like, so long as it and its subclasses properly work with our provided game framework. For example, if our provided code calls your Actor's GetX() function, it should return the correct X value of the actor.

To be compatible with our classes, you ***must*** implement the following public methods in an Actor base class:

- *int GetX( )* and *int GetY( )*: Returns the X and Y coordinates of the Actor as an integer.
- *void SetX(int newXCoord)* and *void SetY(int newYCoord)*: Sets the X and Y coordinates of the Actor.
- *void DoSomething()*: This method is called to allow the Actor to make its move.
- *colors GetDisplayColor( )*: This method returns the color that should be used by the Maze class to display the Actor on the screen during the next tick.
- *int getMyID()*: This ***important*** function should return a numerical value indicating the ID of each Actor (IDs are defined in our provided *constants.h*):
    - Inky must return an ID of ITEM_MONSTER1
    - Stinky must return an ID of ITEM_MONSTER2
    - Dinky must return an ID of ITEM_MONSTER3
    - Clyde must return an ID of ITEM_MONSTER4
    - NachMan must return an ID of ITEM_NACHMAN

Your *DoSomething()* function for each of your Actor objects will be called during each tick of the game. Of course, the *DoSomething()* function for NachMan will be different than the *DoSomething() function* of Inky, Stinky or Clyde. (You'll write code later to call the *DoSomething()* functions for all actors during each tick of the game).

You may implement any other public/private methods you like in your Actor class. You may also add any private data/member variables you like to make your life easier. Remember, since the Actor class is the base class for all of your game's actors, it should contain all functionality and data which is common to all of its subclasses.

Note: Instances of your Actor classes (e.g., NachMan, Monster, Dinky, etc…) don't need to worry about plotting themselves to the screen. They simply need to update their X,Y coordinates and our graphics system will take care of all the gory details of displaying each Actor on the screen!

The NachMan Class

Your NachMan class will be an extension of your Actor class and have a number of additional functions and data to implement NachMan's capabilities. For instance, a NachMan has a score, a number of lives, a current direction he's facing, etc., so you'll have to specialize your NachMan class with these items. In addition, since all Actors are asked to do something during each tick of the game, your NachMan class must have a *DoSomething( )* function (call this function whatever you like) that causes your NachMan to do something during each tick of the game.

NachMan Initialization

- A NachMan always starts out in a specified X,Y location (specified during construction).
- A NachMan starts out with three lives.
- A NachMan starts out facing no particular direction (a direction of *none*) and should not move once the game starts until the player specifies a valid direction. You can find the list of valid directions (*NORTH*, *SOUTH*, *EAST*, *WEST*, *NONE*) defined in our GraphManager.h header file, which you may include in your code.

When the NachMan is Asked to Do Something

Your NachMan class should have a *DoSomething( )* function that can be called by your main game loop during each tick of the game to ask it to do something. This function ***must*** follow these specifications.

- It must move NachMan throughout the maze in the player-selected direction, one square each time the function is called (unless NachMan is not facing a direction, or is stuck trying to move into a wall or the cage door, in which case NachMan should not move). See the section below on movement for more information.
- It must update the player's score and play the appropriate sound when NachMan eats regular pellets, power pellets, and vulnerable monsters. If NachMan moves onto a square with a pellet, your function should remove the pellet from the maze so it can't be eaten twice during the game.
    - Every time NACHMAN eats a regular pellet, it should play a PAC_SOUND_SMALL_EAT sound effect using our provided SoundFX class.
    - Every time NachMan eats a power pellet, it should play a PAC_SOUND_BIG_EAT sound effect using our provided SoundFX class.

- It must update the state of all monsters in the *normal* or *vulnerable* state from their current state to a *vulnerable* state when NachMan eats a power pellet. Note: Only monsters that are in a *normal* or *vulnerable* state at the time NachMan eats the power pellet should have their state changed to a *vulnerable* state (see the formula in the section below). Those monsters in the *monsterdie* or *returntohome* states (see Monsters section below) should not have their state changed to *vulnerable* when NachMan eats a power pellet.
- It must detect if NachMan runs into a monster (i.e. NachMan moves onto a square that a monster occupies at the same time during the current tick):
    - If NachMan runs into a *vulnerable* monster then NachMan eats the monster. In this case, you must increment the player's score by 1000 points and update the eaten monster's state to *monsterdie* so it's no longer vulnerable and it begins the process of returning to its home space in the maze.
    - If NachMan runs into a *normal* (non-vulnerable) monster, then update NachMan's state from *alive* to *dead*. (Your main game loop can then determine that NachMan died and stop the game-play to check if the game is over or if the player gets another chance)
    - If NachMan runs onto the same square as a monster that's in the *returntohome* or *monsterdie* states, then nothing special should happen to NachMan or the monster due to this interaction.

As mentioned in the introduction, each time NachMan eats a regular pellet in the maze (i.e. walks onto the same square as a pellet), your *DoSomething( )* function should give the player 10 additional points.

When NachMan eats a power pellet, he receives 100 points *and* all monsters that are not currently in the *returntohome* or *monsterdie* states are placed into the *vulnerable* state for the following number of game ticks (a game tick is a single iteration of the game in which NachMan's *DoSomething( )* function and all of the monster's *DoSomething( )* functions are called, giving each of the 5 actors a chance to do something):

$$\text{nVulnerableTicks} = \begin{cases} 100 - \text{nLevel} * 10 & \text{if nLevel} <= 8 \\ 20 & \text{if nLevel} > 8 \end{cases}$$

where nLevel is the current maze level of the game (with the starting level of the game being level 0 – you can determine the current level # by calling to the *World::GetLevel()* method). Thus, the minimum amount of time that a monster will be *vulnerable* after NachMan eats a power pellet is 20 ticks. If a monster is currently in the *normal* state and NachMan eats power pellet, the monster must be switched to the vulnerable state for the specified number of ticks. If a monster is already in the *vulnerable* state and then NachMan eats another power pellet, then that monster will have its number of vulnerability ticks reset to the maximum value using the equation above and it will remain in the *vulnerable* state. However, those monsters that are in the process of returning to their start square after being eaten (e.g., in the *monsterdie* or *returntohome* states) will not be placed in a vulnerable state when NachMan eats a power pellet.

For example, if NachMan is on level 0 and just ate a power pellet 15 turns ago, then all monsters not returning to their cage would be vulnerable for 85 further ticks. If, during the 15th tick, NachMan eats another power pellet, then all monsters not returning to their cage in a *returntohome* state or in a *monsterdie* state would be reset to be vulnerable for the full 100 ticks.

Movement

To determine what direction NachMan is facing, your `DoSomething( )` function should prompt the player for a keystroke (we'll tell you how to prompt them below):

- `ARROW_LEFT turns NachMan to face west`
- `ARROW_RIGHT turns NachMan to face east`
- `ARROW_UP turns NachMan to face north`
- `ARROW_DOWN turns NachMan to face south`

Hitting one of the arrow keys sets NachMan's current direction. Once provided with a direction, during each subsequent tick (including the same tick that the player hit the directional key), NachMan will move one square in the specified direction within the maze until he runs into a wall or into a cage door. Hint: Your NachMan class needs to remember its current direction at all times, so that each time its *DoSomething()* function is called, it knows where to move.

NachMan can move one square per tick of the game (i.e. per call to its *DoSomething()* function). Thus, the player needs to hit a directional key only once and NachMan should continue to move, one square each tick, until it runs into an obstruction. Once NachMan hits an obstruction, it should stop until the player specifies another direction where NachMan can legally move**. If NachMan is currently facing a given direction, and the player hits a key that would take NachMan in a different direction directly into an obstruction, then NachMan's current direction should not be changed; instead, NachMan should continue in its original direction.**

Since NachMan is a "real-time" game, you can't use the typical getline and operator>> functions to get player input within the NachMan's DoSomething function. These functions will stop your program and wait until the player types in the proper data and hits the **enter** key. This would make for a really boring NachMan game (requiring the player to repeatedly hit a directional key then hit enter). Instead, you will need to use a special function we provide called `getCharIfAny()` found in our provided header file *UserInterface.h* to get input from the player. This function rapidly checks to see if the player hit a key and returns an indication of whether a key was hit. If the player hit a key, then the key pressed is sent back to the caller of the function. Otherwise, the function immediately returns a result informing the caller that no key was hit. This function could be used as follows:

```
#include "UserInterface.h"
…
```

```
void DoSomething()
{
    char ch;

    if (getCharIfAny(ch))
    {
       switch (ch)
       {
         case ARROW_LEFT:
           SetNachManCurrentDirectionTo(west);
           break;
         case ARROW_RIGHT:
           SetNachManCurrentDirectionTo(east);
           break;

          // etc…
       }
    }

    MoveNachManInCurrentDirection();
    ...
}
```

Mandatory Methods in your NachMan Class

To be compatible with the existing World and Maze classes, your NachMan class *must* have the following public functionality (implemented in a base class when appropriate):

- A constructor that is compatible with our World class. When constructing a new NachMan, our world class passes in a pointer to itself (a pointer to the game's World object), as well as the starting X,Y coordinates of the NachMan.
- *`int GetNumLivesLeft( )`*: Returns how many lives NachMan has left. NachMan starts the game with 3 lives.
- *`void DecrementNumLives( )`*: Decrements the number of lives that NachMan has left. (When the number of lives reaches zero, the World object will end the game.)
- *`int GetScore( )`*: Returns the current score of NachMan. The score should start out at zero at the start of the game.
- *`int GetX( )`* and *`int GetY( )`*: Gets the X and Y coordinates of NachMan.
- *`void SetX(int x)`* and *`void SetY(int y)`*: Sets the X and Y coordinates of NachMan.
- *`colors GetDisplayColor()`:* This method returns the that color that should be used by the maze class when displaying NachMan in the maze. It should always be YELLOW for NachMan. c*olors* is an

enumerated type - all of the *colors* constants (e.g., RED, YELLOW, etc.) can be found in our provided BGIgraphics.h file.

You may implement any additional public and private methods, as appropriate, in your NachMan class.

The Monster Class

Your Monster class and its descendants will be an extension of your Actor class and have a number of additional functions and data items to implement the capabilities of monsters. For example, monsters have a current state they're in (*vulnerable*, *normal*, *monsterdie*, *returntohome*), a current color (which varies depending on what state the monster is in), etc. so you'll have to specialize your Monster class with these items. In addition, since all Actors are asked to do something during each tick of the game, your Monster classes must have a *DoSomething( )* function that causes your Monsters to do something during each tick of the game.

The *DoSomething( )* method in your **Monster** class is responsible for the following:

- Moving each monster, according to its specified algorithm and its current state each time it's asked to do something during a tick. Each monster's specific behavioral algorithm is detailed in the sections below.
- Each monster must maintain its current state. Each monster should be in one of four states at all points during the game:
    - **normal**: the monster is dangerous to NachMan and if it lands on the same square as NachMan, it will kill NachMan. The *normal* state is the initial state of all monsters when starting/restarting a level.
    - **vulnerable**: the monster is vulnerable and may be eaten by NachMan when in this state. If a monster is eaten by NachMan (i.e. they're on the same square), NachMan gets points and the monster switches to the *monsterdie* state. When a monster is in the vulnerable state, it maintains a counter that indicates how many ticks it will stay in this vulnerable state. Once the monster runs out of vulnerable ticks, it immediately transitions back to a *normal state.*
    - **monsterdie:** When the monster is in this state, it should play a PAC_SOUND_BIG_EAT sound effect using our provided SoundFX class (since it was just eaten), and then transition to the *returntohome* state. A monster in the *monsterdie* state should do nothing except play the sound effect and transition to the *returntohome* state when asked to do something during a tick.
    - **returntohome**: When a monster is in this state, it means that it was just eaten by NachMan and is now returning to its home square within the monster cage. A monster in this state can't be eaten and

will simply continue to move until it reaches its home square, at which point it immediately reverts back to the *normal* state.

- When a monster is in the vulnerable state, it is responsible for decrementing its vulnerability count each time its *DoSomething( )* method is called. Once the vulnerability count reaches zero, the monster should switch itself back to a *normal* state.
- When in the *returntohome* state, the monster must move to its home square, moving a single square through the maze each time its *DoSomething( )* function is called. Once the monster reaches its home square, the monster is responsible for setting its state back to *normal*, and the monster again becomes dangerous to NachMan. Monsters must take an ***optimal*** path back to their home position once they are eaten by NachMan (you're going to have to write code to identify this optimal path - see the MyMaze section below for more details).
- Each monster must detect if/when it runs into NachMan:
    - If a monster runs into NachMan while the monster is in a *vulnerable* state, then it must update the player's score and update the monster's state to *monsterdie* so it's no longer vulnerable and will evetually return to its home position in the maze.
    - If the monster runs into NachMan while the monster is in a *normal* state, then it must update the NachMan object to indicate that NachMan has died. Your main game loop can then detect that NachMan has died before the next tick begins, and either end the game (if NachMan has run out of lives) or decrement the number of NachMan lives, then prompt the player, and allow them to continue playing the current level where they left off.
    - If the monster runs into NachMan while it's in the *returntohome* or *monsterdie* states, then the monster should ignore this and continue moving to its home position in the maze.

You may implement any public/private methods, as appropriate, in your Monster class – you should make every effort to factor out all common behavior from your monsters and place it in this class, so you don't duplicate logic across multiple classes.

<u>Monster Movement while in a Normal or Vulnerable state</u>

All monsters use the same basic movement algorithm while in a *normal* or *vulnerable* state. During each tick of the game (e.g., during a call to its *DoSomething( )* function) each monster decides upon a particular square in the maze it would ultimately like to move to. Each monster (e.g., Inky vs. Stinky) uses a different algorithm to decide what square is its destination – these algorithms are described in each Monster's subsection below. Once a monster determines which x,y coordinate (a location of 0,0 represents the upper-left corner of the maze) in the maze is its latest destination, it must then move one square in one of the four directions (north, south, east or west) to try to get closer to that destination. Video games look really crummy if monsters constantly move back and forth like they're confused, so one feature of our movement algorithm below, is that a monster will never intentionally reverse its course unless it absolutely has to.

Each monster ***must*** use the following algorithm to decide which direction to move in to best reach its target square when it's in a *normal* or *vulnerable* state. The monster must:

1. Determine whether the target destination is east, west or in the same column as the Monster.
2. If the monster's destination is in a different column than the monster (the destination is somewhere east or west of the monster), then:
    a. Check if the monster can move horizontally one square closer to the destination without running into a wall AND
    b. Check whether or not moving in this east/west direction would cause the monster to directly reverse its course from the last tick. For example, if, during the last turn, the monster had moved west, and now the monster decides that its target location is eastward, then moving eastward would cause it to reverse its direction.
    c. If the monster can move in the desired horizontal direction AND in doing so, it will not be reversing its last movement direction, then the monster should set its x coordinate to be one horizontal square closer to its destination. This is the only movement it can make during its turn (the monster can still do other things like check to see if it moved onto a square with NachMan, for example, but it can't move vertically during this tick).
3. Otherwise (if the Monster didn't attempt to move east or west), determine whether the target destination is north, south or in the same row as the monster.
4. If the monster's destination is in a different row than the monster (the destination is somewhere north or south of the monster), then:
    a. Check if the monster can move vertically one square closer to the destination without running into a wall AND
    b. Check whether or not moving in this north/south direction would cause the monster to directly reverse its course from the last tick. For example, if, during the last turn, the monster had moved north, and now it decides that its target location is southward, then moving southward would cause it to reverse its direction.
    c. If the monster can move one square the desired vertical direction AND it will not be reversing its last movement direction, then the monster should set its y coordinate to be one horizontal square closer to its destination. This is the only movement it can make during its turn (the monster can still do other things like check to see if it moved onto a square with NachMan, for example, but it can't move any more during this tick).
5. Otherwise, if the monster couldn't find a direction to move in steps 1-4 above, the monster can't move directly toward its target, so it must pick a random direction D (north, south, east or west) to move and then…
6. Repeat four times:
    a. If the monster can move in direction D (e.g., D = north) without running into a wall, and such movement would not cause the monster

to reverse its direction from the last tick, then move the monster one square in direction D. This is the only movement the monster can make during its turn (the monster can still do other things like check to see if it moved onto a square with NachMan, for example, but it can't move any more during this tick).

b. Otherwise, advance to the next possible direction (e.g., from north to south, from south to east, from east to west, or from west to north) and see if this new direction is a valid option.

7. If none of the above options were satisfied, then it means that the monster cannot move without reversing its course. In this case, the monster must move one square in the opposite direction of its movement during the last tick of the game.

Monster Movement while in a ReturnToHome state

When your monster is in the *returntohome* state, its only goal is to return to its home square in the maze in the most efficient manner possible. Therefore, while in this state, during each tick, it *must* move one square in a direction that takes it one step closer to its home square in the maze (this could require the monster to temporarily move away from its home square if a wall, for example, prevents the monster from directly moving toward the home square in an optimal fashion). During each subsequent tick, the monster must continue to move exactly one square until it reaches its starting position. The monster must not move through walls, but may move on all other squares in the maze to reach its home square.

In order to determine the optimal path from the monster to its home square, your monster classes will leverage a new class called MyMaze (that you will write) that is derived from our Maze class. Your job is to create a valid MyMaze class which is a subclass of our provided Maze class, and add algorithms/data structures to this subclass that can be used by a monster to help it find its way from any square in the maze to its home square. To do this, you *may* subclass any of the *virtual* functions in the Maze class, as well as adding your own constructor. You may also add any additional private data structures and algorithms that you need in your derived class. You *must* also write the implementation for the *GetNextCoordinate( )* function in your MyMaze class. This function will be called by your monsters, while they're in a *returntohome* state, to determine the best possible, *legal,* newX, newY coordinate to move to based on the monster's current curX and curY coordinate.

```
class MyMaze
{
public:
   ...
   bool GetNextCoordinate(int curX, int curY, int &newX, int &newY);
   ...
};
```

For more information on the GetNextCoordinate function, please see the MyMaze section below.

The Monster Subclasses

This section describes all of the detailed requirements for your Monster subclasses: Inky, Stinky, Dinky and Clyde. What you'll notice is that many of these monsters share similar behaviors. Therefore, where feasible, you should place common behaviors/algorithms in your base Monster class, rather than re-code the same thing over and over in your Monster subclasses. **If we find that you have added substantially similar functions to each of your subclasses that could otherwise be generalized and placed into your Monster base class, you will lose points!**
Your Inky, Stinky, Dinky and Clyde class must be usable with all of our provided classes without requiring modification to any of our classes. For example, our World class instantiates new Monsters like this:

```
// initialize our monsters
m_apcMonster[0] = new Inky(this,m_pcMaze->GetMonsterStartX(),m_pcMaze->GetMonsterStartY());
m_apcMonster[1] = new Stinky(this,m_pcMaze->GetMonsterStartX(),m_pcMaze->GetMonsterStartY());
m_apcMonster[2] = new Dinky(this,m_pcMaze->GetMonsterStartX(),m_pcMaze->GetMonsterStartY());
m_apcMonster[3] = new Clyde(this,m_pcMaze->GetMonsterStartX(),m_pcMaze->GetMonsterStartY());
```

Therefore, you must make sure your Inky, Stinky, Dinky and Clyde subclasses have constructors that accept a World pointer (for the first parameter), and X and Y starting coordinates for the second and third parameters.

In general, make sure that your Monster subclasses work with our provided World and Maze classes as-is! ***You must not modify our provided classes to make them work with your classes!***

Inky Details

Your **Inky** class *must* use the following algorithms (within the *DoSomething( )* method) to decide how to move, in each of his states. Inky *must m*ove one square each time its *DoSomething( )* method is called unless it is in the *monsterdie* state.

Normal State

1. When Inky is first initialized (and any time Inky transitions back into a normal state), he has to decide if he wants to chase NachMan for a while or wants to just cruise the maze for a while (sometimes Inky feels wild and crazy and wants to cruise).
2. When it's time for Inky to decide what to do (every 10 ticks), his code needs to generate a random number V between 0 and 99.
3. If 0 <= V < 80, then Inky will start chasing NachMan for exactly 10 ticks (e.g., there's an 80% chance that Inky will want to chase NachMan).
4. If 80 <= V < 100, then Inky will cruise the maze for exactly 10 ticks, not intentionally trying to chase NachMan.

5. After the 10 ticks have elapsed of either behavior, Inky decides once again if he wants to chase NachMan or just cruise the maze (i.e., he generates a random number again and resets his count to 10, etc).
6. If Inky is currently interested in chasing NachMan and his 10 ticks haven't elapsed, then during each tick he will set his *target coordinate* to NachMan's current coordinate (Inky is psychic and knows where NachMan is in the maze even if he can't directly see him). During the same tick, he then uses the movement algorithm described in the Monster section to move one square toward his target.
7. If Inky is not currently interested in chasing NachMan, then during each tick he will set his *target coordinate* to a random x,y coordinate in the maze (it does not matter whether this target coordinate is actually reachable). During the same tick, he then uses the movement algorithm described in the monster section to move one square toward this target.
8. Regardless of whether Inky is interested in chasing NachMan, at the end of every tick (after moving itself to a new square) he needs to check to see if he has landed on the same square as NachMan. If so, and Inky is in a *normal* state then Inky should set NachMan's state to dead so the main game loop knows to stop the current round and tell the player that NachMan has died.

Vulnerable State

If Inky is in a *vulnerable* state, then during *each* tick he will set his target coordinate to a random x,y coordinate in the maze (it does not matter whether this target coordinate is actually reachable). During the same tick, he then uses the movement algorithm described in the Monster section to move one square toward this target. The target square therefore changes during every single tick while Inky is in a *vulnerable* state.

During each tick while being *vulnerable*, Inky should decrement his vulnerable count until it reaches zero. When this count reaches zero, Inky then reverts back to a *normal* state (where he'll use the movement algorithm described above).

At the end of every tick (after moving itself to a new square), Inky needs to check to see if he has landed on the same square as NachMan. If so and Inky is in a *vulnerable* state then Inky should award the player 1000 points (for eating Inky) and then set his own state to *monsterdie*. The *DoSomething( )* function should then return without doing anything else.

MonsterDie State

When Inky transitions into a *monsterdie* state from the *vulnerable* state, he should make a PAC_SOUND_BIG_EAT noise using our sound effects class.

Once Inky is in the *monsterdie* state, during the next call to his *DoSomething( )* function, he simply transitions to the *returntohome* state but does *nothing else*. In other words, while in this state, Inky won't move during the current tick. He will

simply change his state and then return, only moving again the next time his *DoSomething( )* function is called once he's in a *returntohome* state.

ReturnToHome State

Inky must move one square toward his start square each time his *DoSomething( )* method is called. See the MyMaze section below for more details on how Inky decides how to move toward his home square. Once Inky returns to his start square, he should set his state to *normal* and proceed with his *normal* behavior.

Clyde Details

Your **Clyde** class *must* use the following algorithms (within the *DoSomething( )* method) to decide how to move, in each of his states. Clyde *must m*ove one square each time its *DoSomething( )* method is called unless it is in the *monsterdie* state.

Normal State

Clyde is a lover, not a fighter, and he hates to chase NachMan. As such, while in the *normal* state, during *each* tick he will set his target coordinate to a random x,y coordinate in the maze (it does not matter whether this target coordinate is actually reachable). During the same tick, he then uses the movement algorithm described in the monster section to move one square toward this target. The target square therefore changes during every single tick.

At the end of every tick (after moving itself to a new square) he needs to check to see if he has landed on the same square as NachMan. If so, and Clyde is in a *normal* state then he should set NachMan's state to dead so the main game loop knows to stop the current round and tell the player that NachMan has died.

Vulnerable State

If Clyde is in a *vulnerable* state, then during *each* tick he will determine which quadrant of the maze NachMan is in (upper left, upper right, lower left, or lower right). Each quadrant is exactly one quarter of the maze (of size MAZE_WIDTH/2 by MAZE_HEIGHT/2). Clyde will then set his target location to the exact opposite corner of the maze from NachMan. For example, if NachMan is anywhere in the upper right quadrant, then Clyde will set his target square to the lower left corner (x=0,y=14). If NachMan were in the lower right quadrant, then Clyde would set his target location to the upper left-hand corner (x=0,y=0). During the same tick, he then uses the movement algorithm described in the monster section to move one square toward this target. The target square therefore changes during every single tick while Clyde is in a vulnerable state.

During each tick while being *vulnerable*, Clyde should decrement his vulnerable count until it reaches zero. When this count reaches zero, Clyde then reverts back to a *normal* state (where he'll revert to the algorithm described above).

At the end of every tick (after moving itself to a new square), Clyde needs to check to see if he has landed on the same square as NachMan. If so and Clyde is in a *vulnerable* state then Clyde should award the player 1000 points (for eating Clyde) and then set his own state to *monsterdie*. The *DoSomething( )* function should then return without doing anything else.

MonsterDie State

When Clyde transitions into a *monsterdie* state from the *vulnerable* state, he should make a PAC_SOUND_BIG_EAT noise.

Once Clyde is in the *monsterdie* state, then he simply transitions to the *returntohome* state but does *nothing else*. In other words, while in this state, Clyde won't move during the current tick. He will simply change his state and then return, moving again the next time his *DoSomething( )* function is called.

ReturnToHome State

Clyde must move one square toward its start square each time its *DoSomething( )* method is called. See the MyMaze section below for more details. Once Clyde returns to his start square, he should set his state to *normal* and proceed with his *normal* behavior.

Stinky Details

Your **Stinky** class *must* use the following algorithms (within the *DoSomething( )* method) to decide how to move, in each of his states. Stinky *must m*ove one square each time its *DoSomething( )* method is called unless it is in the *monsterdie* state.

Normal State

Stinky has a very keen sense of smell while he's hunting in his *normal* state. As such, if Stinky is within 5 vertical squares away of NachMan ***and*** within 5 horizontal squares away from NachMan, he will set his target coordinate to the coordinate of NachMan.

For example, if NachMan is at location x=10,y=10 and Stinky is at location x=15,y=14, then Stinky will smell NachMan and set his target location to x=10,y=10. On the other hand, if Stinky were at x=16, y=10, Stinky would be too far away from NachMan (at x=10,y=10) to smell him.

If, on the other hand, Stinky is not within this 5x5 range of NachMan then he can't smell him. In this case he simply picks a random x,y coordinate in the maze (it does not matter whether this target coordinate is actually reachable) as his target location.

During the same tick, he then uses the movement algorithm described in the monster section to move one square toward his chosen target. The target square therefore changes during every single tick.

At the end of every tick (after moving itself to a new square) he needs to check to see if he has landed on the same square as NachMan. If so, and Stinky is in a *normal* state then he should set NachMan's state to dead so the main game loop knows to stop the current round and tell the player that NachMan died.

Vulnerable State

If Stinky is in a *vulnerable* state, then during *each* tick he will set his target coordinate to a random x,y coordinate in the maze (it does not matter whether this target coordinate is actually reachable). During the same tick, he then uses the movement algorithm described in the monster section to move one square toward this target. The target square therefore changes during every single tick while Stinky is in a *vulnerable* state.

During each tick while being *vulnerable*, Stinky should decrement his vulnerable count until it reaches zero. When this count reaches zero, Stinky then reverts back to a *normal* state (where he'll revert to the algorithm described above).

At the end of every tick (after moving itself to a new square), Stinky needs to check to see if he has landed on the same square as NachMan. If so and Stinky is in a *vulnerable* state then Stinky should award the player 1000 points (for eating Stinky) and then set his own state to *monsterdie*. The *DoSomething( )* function should then return without doing anything else.

MonsterDie State

When Stinky transitions into a *monsterdie* state from the *vulnerable* state, he should make a PAC_SOUND_BIG_EAT noise.

Once Stinky is in the *monsterdie* state, then he simply transitions to the *returntohome* state but does *nothing else*. In other words, while in this state, Stinky won't move during the current tick. He will simply change his state and then return, moving again the next time his *DoSomething( )* function is called.

ReturnToHome State

Stinky must move one square toward its start square each time its *DoSomething( )* method is called. See the MyMaze section below for more details. Once Stinky returns to his start square, he should set his state to *normal* and proceed with his *normal* behavior.

<u>Dinky Details</u>

Your **Dinky** class *must* use the following algorithms (within the *DoSomething( )* method) to decide how to move, in each of his states. Dinky *must m*ove one square each time its *DoSomething( )* method is called unless it is in the *monsterdie* state.

Normal State

Dinky has excellent vision. As such, if Dinky is in a horizontal or vertical line of sight of NachMan (and there are no ***walls*** in the way), he will set his target location to the coordinate of NachMan. If, on the other hand, Dinky cannot see NachMan then he just forgets about him. In this case he simply picks a random x,y coordinate in the maze (it does not matter whether this target coordinate is actually reachable) as his target location.

During the same tick, he then uses the movement algorithm described in the monster section to move one square toward his chosen target. The target square therefore changes during every single tick.

At the end of every tick (after moving itself to a new square) he needs to check to see if he has landed on the same square as NachMan. If so, and Dinky is in a *normal* state then he should set NachMan's state to dead so the main game loop knows to stop the current round and tell the player that NachMan died.

Vulnerable State

If Dinky is in a *vulnerable* state, then during *each* tick he will set his target coordinate to a random x,y coordinate in the maze (it does not matter whether this target coordinate is actually reachable). During the same tick, he then uses the movement algorithm described in the monster section to move one square toward this target. The target square therefore changes during every single tick while Dinky is in a *vulnerable* state.

During each tick while being *vulnerable*, Dinky should decrement his vulnerable count until it reaches zero. When this count reaches zero, Dinky then reverts back to a *normal* state (where he'll revert to the algorithm described above).

At the end of every tick (after moving itself to a new square), Dinky needs to check to see if he has landed on the same square as NachMan. If so and Dinky is in a *vulnerable* state then Dinky should award the player 1000 points (for eating Dinky) and then set his own state to *monsterdie*. The *DoSomething( )* function should then return without doing anything else.

MonsterDie State

When Dinky transitions into a *monsterdie* state from the *vulnerable* state, he should make a PAC_SOUND_BIG_EAT noise.

Once Dinky is in the *monsterdie* state, then he simply transitions to the *returntohome* state but does *nothing else*. In other words, while in this state, Dinky won't move during the current tick. He will simply change his state and then return, moving again the next time his *DoSomething( )* function is called.

ReturnToHome State

Dinky must move one square toward its start square each time its *DoSomething( )* method is called. See the MyMaze section below for more details. Once Dinky returns to his start square, he should set his state to *normal* and proceed with his *normal* behavior.

Maze and MyMaze

You will be provided with a Maze class that is capable of loading and displaying mazes on the screen. You *must not* modify the Maze class – you must use it as is. You are required to create a new class called MyMaze which is a subclass of the Maze class.

As you may recall from above, one of the project requirements is that once NachMan eats a vulnerable monster (i.e. moves onto the same square as a monster) or a vulnerable monster runs onto the same square as NachMan, the eaten monster's state will change to a *monsterdie* state, and then to a *returntohome* state where each time the monster is given a chance to move, it must move one square closer to its home square until it reaches its home square. The monster must not move through walls, but may move on all other squares in the maze to reach the cage.

Your job is to create a MyMaze subclass containing logic that can help a monster find its way from anywhere in the maze to its home square. To do this, you will implement the *GetNextCoordinate( )* member function in your new MyMaze class. You may also subclass any of Maze's virtual functions, create a constructor/destructor, and add your own private methods and variables to MyMaze. The signature of the *GetNextCoordinate* function is as follows:

```
bool GetNextCoordinate(int curX, int curY, int& newX, int& newY);
```

Given the current x,y coordinate of a monster, this function will determine the next coordinate that the monster should move to in order to reach its home square. If the monster is already on its home square (curX == homeX and curY == homeY when the function is called), then the function must return false and newX and newY must not be changed. Otherwise, if curX and curY are not the same as the home square, then newX and newY must be set to the coordinates the monster must move to. Given that monsters may move only one step at a time north, south, east, or west, the function must set one of the two variables, newX or newY, to the same value as curX or curY, respectively, and the other to a value that differs by 1 from its corresponding curX or curY.

Recall the stack-based algorithm in Homework #2. This algorithm computed the distance values for each square in a maze to some destination square. Place this algorithm in one

or more private member functions of your MyMaze class. Define a new version of the LoadMaze function in your MyMaze class and have it use your new function(s) to compute the distance values for every square in the maze to the monsters' home square every time a new maze is loaded. Store your distance matrix in a separate private two-dimensional array member variable within your MyMaze class. Here's an example maze. The '$' character indicates the monsters' home square:

```
####################
#*................*#
#.##.#.#######.#.##.#
#....#........#....#
####.#.###%###.#.####
#......#  $  #......#
####.#.#######.#.####
#  #.#....@....#.#  #
####.#.#######.#.####
#........#........#
#.##.####.#.####.##.#
#*.#............#.*#
##.#.#.#######.#.#.##
#..................#
####################
```

Your algorithm should yield an array containing the following numbers:

```
99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99
99  13  12  11  10   9   8   9  10  11  12  11  10   9   8   9  10  11  12  13  99
99  14  99  99  11  99   7  99  99  99  99  99  99  99   7  99  11  99  99  14  99
99  15  14  13  12  99   6   5   4   3   2   3   4   5   6  99  12  13  14  15  99
99  99  99  99  11  99   7  99  99  99   1  99  99  99   7  99  11  99  99  99  99
99  13  12  11  10   9   8  99   2   1   0   1   2  99   8   9  10  11  12  13  99
99  99  99  99  11  99   9  99  99  99  99  99  99  99   9  99  11  99  99  99  99
99  99  99  99  12  99  10  11  12  13  14  13  12  11  10  99  12  99  99  99  99
99  99  99  99  13  99  11  99  99  99  99  99  99  99  11  99  13  99  99  99  99
99  17  16  15  14  13  12  13  14  15  99  15  14  13  12  13  14  15  16  17  99
99  18  99  99  15  99  99  99  99  16  99  16  99  99  99  99  15  99  99  18  99
99  19  20  99  16  17  18  19  18  17  18  17  18  19  18  17  16  99  20  19  99
99  99  21  99  17  99  19  99  99  99  99  99  99  99  19  99  17  99  21  99  99
99  21  20  19  18  19  20  21  22  23  24  23  22  21  20  19  18  19  20  21  99
99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99
```

You will recall that in the homework, a value of 99 indicates a wall or an unreachable square (one that is completely walled off). A value of zero indicates the home square, since it is zero squares away from itself.

Using this pre-computed array, you can now write your GetNextCoordinate function. Given the curX,curY coordinate of a monster attempting to return to its starting position, the function should look at distance_grid[curY][curX] to obtain the distance value there. Next, the monster should look in the distance grid at all adjacent squares to the north, south, east and west. The function should return the coordinates of the adjacent square whose number is exactly one less than the monster's current square, since that square is exactly one square closer to the monster's starting position. If two adjacent squares are

equidistant from the starting square, then the function may choose either of the two squares to move to. So for example, consider the maze above. If the monster was at x=6,y=1, its four neighbor squares are (5,1), (7,1), (6,0), (6,2). Since square (6,0) is a wall, this square can be ignored. The square to the south has a value of 7, which is one less than the distance of the current square, which is 8. Therefore, our function would return (6,2) as the next move for our monster. If a monster is on a square whose distance is 0 from the starting position, this means that the monster is already on the starting square and the function should return false without changing its parameters.

Each Maze will be loaded from a text file using the Maze class provided with this project. You should try creating your own mazes to make sure your new functions work as expected. Here is how to define a maze (you can look at the sample mazes too):

1. Each maze must be exactly 21 squares wide by 15 squares high.
2. Use the following characters in the maze file:
    a. # to represent a wall
    b. . (period) to represent a food pellet
    c. * to represent a power pellet
    d. A space to represent an empty square
    e. A % to represent a monster cage door (that only monsters can move through)
    f. A $ to represent the starting position for all of the monsters when the level starts. The $ must be in an enclosed space of the maze, and the enclosed space must have at least one monster door to enable the monsters to exit their cage. There may only be one $ in each maze.
    g. An @ sign specifies where the NachMan character should start in the maze at the start of a level or when NachMan loses dies but has more tries left to play. There may only be one @ in each maze.
3. All mazes *must* have walls completely surrounding the outside of the maze.

World Class Details

The World class, which we provide, is the class in the project that brings everything together: it contains the current maze and it holds NachMan and all of the monster objects. It's responsible for loading each maze/level of the game, initializing all of the monsters and NachMan in preparation to play, and then calling a function that you'll program, called *RunLevel( ),* which is responsible for running the actual game-play until either NachMan dies or the player finishes the current level. Once the current level is completed, the World class then advances to the next level, loads the next maze, etc. Or, if NachMan dies before completing the level, the world class has the logic to give the player another chance or to end the game if NachMan runs out of lives.

We provide the entire World class *except* for a single function, called *RunLevel( )* which is a pure-virtual function. You must define a subclass of World, called MyWorld, and implement the *RunLevel( )* function in it. You may add additional private member functions, as required, to your MyWorld class, but you must not add any additional

public member functions. You must not modify our World class in any way. In other words, all of your other classes (Actor, NachMan, Monster, etc.) must only interact with the World/MyWorld classes using the public member functions we have already defined for you.

What must your RunLevel function do? Here's the exact pseudo-code for it that you must use to implement your function:

```
GameStatus MyWorld::RunLevel()
{
      // On entry to this function, the NachMan, Monster and
      // Maze objects have already been created. To get a pointer
      // to them, you can use the World class's GetNachMan(),
      // GetMonster(), and GetMaze() methods.

      // YOU MUST WRITE CODE TO DO EXACTLY THIS:

      // NachMan initialization:
      // 1. Set NachMan's state to ALIVE and
      // 2. Set NachMan's current location to the
      //    specified NachMan-start-location in the Maze
      // 3. Reset the NachMan's direction to none so he
      //    doesn't move until the player hits a key

      // Monster initialization:
      // 1. Set all four monsters' states to normal
      // 2. Set each monster's start location in the maze
      //    to the monster home square in the Maze

      // Redisplay every cell on the screen before
      // game play begins (true means redraw every grid cell)
      DisplayScreen(true);  // use this exact function call

      // Define your main game loop
      // Each iteration of the loop is one game tick

      while (NachMan-is-not-dead && there-are-uneaten-pellets-in-maze)
      {
            // Ask the NachMan object to move itself

            // Check if NachMan has died as a result of
            // moving itself.  If NachMan has not died,
            // then ask each of the four monsters to move itself

            // Now update the graphical display, only redrawing
            // those sqares on the grid that changed due to
            // movement (That's what the false means)
            DisplayScreen(false); // use this exact function call
      }

      if (all of the food was eaten)
```

```
	{
		// you must return finishedlevel if NachMan
		// has eaten all of the pellets on this level
		// and therefore has completed the level
		return finishedlevel;
	}

	// otherwise, NachMan died; you must return playerdied
	// in this case.
	return playerdied;
}
```

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 iteratively almost always fail 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 getMyID() { return -1; }    // 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, re-compile 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.

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 or thousands of errors and just get frustrated! So don't do it.

Project Setup

Here's what you should do to set up this project under Visual Studio:

1. Create a new project called, say, Nachenroids.
2. Extract the contents of the nachman-skeleton.zip file into the folder that already contains the .vcproj file. If your project is named Nachman, this will be the folder Nachman/Nachman.
3. In the Solution Explorer window, right-click Header Files, and use Add Existing Item to add all the .h files.
4. In the Solution Explorer window, right-click Source Files, and use Add Existing Item to add all the .cpp files.
5. In the Project menu, select Nachman Properties… . On the Nachman Property Pages window that pops up, click the + to expand the Configuration Properties, then the + to expand Linker. Under Linker, select Input. In the pane that appear to the right, select Additional Dependencies. In the box to its right, select the dropdown list choice <Edit>, type `winmm.lib` in the upper pane of the Additional Dependencies dialog box, and click OK and then OK again. (This step ensures that calls to SoundFX::PlaySound will resolve to code in the Windows multimedia library.)

What To Turn In

Part #1 (30%)

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 NachMan game that implements maybe 20% of the overall project. You must program:

1. An Actor base class for all of your actors:
    i. It must have a simple constructor and destructor.
    ii. It must have a single function called *DoSomething()* that can be called by the world to get an Actor to do something.
    iii. It must define *GetX(), GetY(), SetX() and SetY()* functions defined.
    iv. It must have a *getMyID()* function defined that simply returns whatever the current Actor's ID# is.

v. It must have a *GetDisplayColor()* function defined that returns the actor's current color.
vi. You may add other public or private functions to your Actor class, as you see fit.

2. A limited version of your NachMan class, derived from Actor:
   i. It must have a simple constructor and destructor that are compatible with our provided World class, so the World can construct your NachMan.
   ii. It must have a limited version of a *DoSomething()* function that lets the player pick a direction by hitting a key, and if there is no wall or cage door in the specified direction, moves the NachMan in that direction. All your *DoSomething()* method has to do is properly adjust the NachMan's X,Y coordinates and our graphics system will automatically animate its movement it around the maze!
   iii. Your NachMan must return its proper object ID number (ITEM_NACHMAN) when its *getMyID()* function is called.
   iv. Your NachMan must return its proper color (YELLOW) when its *GetDisplayColor()* function is called.
   v. Your NachMan should have some function to determine if it's dead or not, which your World::*RunLevel()* function can use to determine when NachMan has died. For part 1, this function can simply return false all the time, since NachMan can't die yet since there are no monsters.
   vi. You may add any set of public and private functions to your NachMan class as you see fit, so long as you use good object oriented programming style (e.g., don't duplicate functionality).
3. A limited version of your *RunLevel()* method in your MyWorld class. You may ignore, for part 1 of this assignment, all items in the pseudocode that refer to monsters. You should program enough of the *RunLevel()* function to be able to move the NachMan around the maze.

Once you've implemented these two classes and the *RunLevel()* method of the MyWorld class, uncomment the following line in the testdefines.h file:

```
#define PROJ3_PART_ONE
```

This #define ensures that every part of our project can now work without the Monster classes – in other words, you can compile our code even though you haven't defined your Monster classes. Our provided code can now work with your simple Actor and NachMan classes, alone. You can now test your NachMan in isolation without having to worry about anything else in the game.

Now compile your program – you'll probably start out with lots of errors… Relax and try to remove all of the compile errors and get your program to run.

You'll know you're done with part 1 when your program compiles and does the following: When it runs, it should display the starting maze of the game and the current level number. NachMan should be in the center of the screen, facing the right, drawn in a YELLOW color. If your Actor and NachMan classes work properly, you should be able to move NachMan around the screen using the arrow keys or the '2', '4', '6' and '8' keys. Your NachMan should only move during the tick when the player hits the directional keys to move.

REMEMBER TO COMMENT OUT THE:

#define PROJ3_PART_ONE

LINE WHEN DOING PART 2 OF YOUR PROJECT!

Your Part #1 solution may actually do more than what is specified above; so for example, if you are further along in the project, and what you have compiles and has at least as much functionality as what's described above, then you can turn that in instead.

Note, the Part #1 specification above doesn't require you to implement monsters (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 NACHMAN CLASS STILL WORKS PROPERLY AND THAT YOUR PROGRAM STILL COMPILES AND MEETS THE REQUIREMENTS STATED ABOVE FOR PART #1!

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

What to Turn In For Part #1

You must turn in your source code for the simple version of your game, which ***compiles without any errors in Visual Studio***. It must consist of these ***exact*** files and no other source files:

```
Actor.h        // contains your Actor and NachMan classes and any
               // constants required by these classes
Actor.cpp      // contains the implementation of your Actor and NachMan classes
MyWorld.h      // contains your MyWorld class definition
MyWorld.cpp    // contains your MyWorld class and your RunLevel() method
```

You may optionally turn in:

```
MyMaze.h       // if you have created your MyMaze class, submit this
MyMaze.cpp     // if you have created your MyMaze class, submit this
```

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!

Part #2 (70%)

After you have turned in your work for Part #1 of Project 3, your TA 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 provided by the TA**.

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

What to Turn In For Part #2

You must turn in the following files, and ONLY the following files. If you name your source files with other names, you will be docked points, so be careful!

```
Actor.h        // contains declarations for all of your Actor classes
Actor.cpp      // contains implementations for all of your Actor classes
MyWorld.h      // contains your MyWorld class definition
MyWorld.cpp    // contains implementation of your MyWorld class
MyMaze.h       // contains the definition of your MyMaze class
MyMaze.cpp     // contains the implementation of your MyMaze class

report.doc     // your report (10% of your grade)
```

You must turn in a report. The report should contain the following:

1. A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual. For example, "I chose to define a pure virtual version of the blah() function in my base Actor class because all Actors have a blah function, and all of them define their own version of it."
2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. "I wasn't able to implement shooting of bullets." or "My Inky doesn't work correctly yet so I just treat it like a Dinky right now."
3. A list of other design decisions and assumptions you made, e.g.:
    i. It was ambiguous what to do in situation X, and this is what I decided to do.
4. A description of how you tested each of your classes (1-2 paragraphs per class)

FAQ

Q: The specification is ambiguous. What should I do?
A: Play with our sample program and do what it does. Use our program as a reference. If the specification is ambiguous and our program is ambiguous, do whatever you like and document it in your report. **If the specification is ambiguous, 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 compiles! If we can sort of play your game, but it's not perfect, that's better than it not even compiling!

Q: Where can I go for help?
A: Try Eta Kappa Nu – they provide free tutoring and can help your with your project!

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

GOOD LUCK!