Project 4
IceMan

For questions about this project, first consult Piazza.

Part 1: 11:59 PM, Tuesday, August 8
Part 2: 11:59 PM, Thursday, August 17

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

BACK UP YOUR CODE REGULARLY TO AN EXTERNAL TARGET LIKE A FLASH DRIVE OR BOX.COM – WE WILL NOT ACCEPT EXCUSES LIKE YOUR COMPUTER BREAKING OR BEING STOLEN.

FINALLY, WE ASK STUDENTS SELF-LIMIT QUESTIONS TO 1 EMAIL PER DAY! IF YOU'RE SOMEONE WITH LOTS OF QUESTIONS, SAVE THEM UP AND ASK ONCE.

Table of Contents

Introduction

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

In Iceman, the player has to dig underground tunnels through an oil field looking for oil. After the player has gathered all available oil within a particular field, the level is completed and the player advances to a new, more difficult oil field.

Here is an example of what the Iceman game looks like:

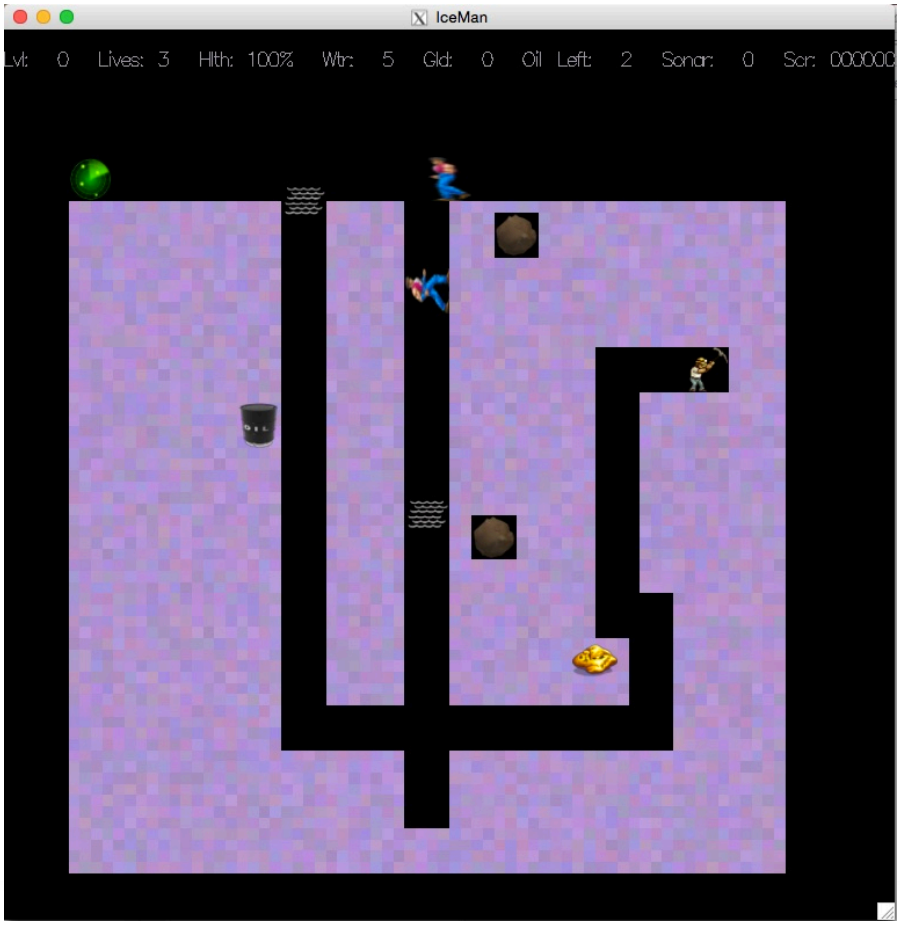

Figure #1: A screenshot of the Iceman game. You can see the *Iceman* (upper-right), two *Hardcore Protesters* in light blue (*Regular Protesters,* here not shown, are in dark blue, and one *Hardcore Protesters* in light blue), several *Boulders*, two *water pools*, a *sonar* kit, a discovered *Barrel* of oil, and a discovered *Gold Nugget*.

Game Details

In Iceman, the player starts out a new game with three lives and continues to play until all of his/her lives have been exhausted. There are multiple levels in Iceman, beginning with level 0, and each level has its own unique oil field configuration. During each level, the *Iceman* (controlled by the player) must dig and find all of the oil hidden within the oil field in order to move on to the next level.

Upon starting each level, the player's *Iceman* avatar (a miner) is placed at the top-middle of the oil field just above the tunnel. Every time the player starts (or re-starts) a level, the *Iceman* is restored to 100% health, given fresh water for their squirt gun (enough for 5 squirts), and given a single sonar charge that they can use to search for nearby items within the ice. Each oil field has a number of *Boulders*, *Gold Nuggets*, and *Barrels* of oil randomly distributed throughout the field. Only the *Boulders* will be visible initially – the other items must be discovered by the *Iceman* before they become visible. The player must use the arrow keys to move the *Iceman* left, right, up and down through the oil field; the *Iceman* digs through any ice in the direction of motion. The *Iceman* may move/dig anywhere in the oil field except for locations occupied by
Boulders.

The *Iceman* must dig through the oil field in search of every *Barrel* of oil and pick each one up by moving to its location. Once the *Iceman* has picked up all of the *Barrels* of oil in the current oil field, the level ends and the player advances to the next level.

Unfortunately, the *Iceman* is not alone. A band of *Regular* and *Hardcore Protesters* roam around the oil field and shout at the *Iceman* any time they get near to him. Each time the *Iceman* is shouted at, he becomes more irritated (and less healthy) and eventually he gives up and quits (loses a life). At this point, if the *Iceman* has more lives, the player can continue the game on the current level. If not, the game is over.

In addition to digging and walking around the oil field, the *Iceman* may also shoot their squirt gun (if it's filled with water) by pressing the space bar. *Squirts* from the squirt gun will irritate the *Protesters* and temporarily stun them. A *Protester* who is sufficiently irritated (their hit points go to zero) will leave the oil field out of frustration. Occasionally, *Water Pools* will randomly bubble up within the tunnels, and the *Iceman* can move onto them to use this water to refill the squirt gun. But beware – a *Water Pool* disappears quickly back into the tunnel's ice, so the *Iceman* needs to siphon it off into the squirt gun quickly lest they disappear.

The *Iceman* will also discover *Gold Nuggets* randomly distributed throughout the oil field. If the *Iceman* picks up one or more gold *Nuggets*, he can use them to bribe the *Protesters* by dropping a *Nugget* in front of a *Protester*. If a *Protester* runs into such dropped gold, it will pick it up and accept it as a bribe. A bribed *Regular Protester* leaves the oil field immediately. However, a *Hardcore Protester* simply pockets the gold and after briefly stopping to gloat over this new treasure, resumes the hunt to protest against

the *Iceman*. If the player drops a *Gold Nugget* and it's not picked up quickly, it will sink into the muck of the oil field, never to be discovered again.

In addition to squirting and bribing the *Protesters*, the *Iceman* can also cause *Boulders* to drop on them. If the *Iceman* digs all of the ice supporting a *Boulder* from underneath it, then after a short delay, the *Boulder* will fall until it crashes into the ice or another *Boulder* below. A *Protester* who is bonked by a *Boulder* as it falls becomes so irritated that it will immediately leave the oil field. But beware, the *Iceman* is also susceptible to being bonked. If the *Iceman* is bonked, he also becomes super irritated (lose all of his hit points), and the level will end. If the player still has lives remaining, the level is restarted.

Of course, the *Iceman* is not alone – he has buddies that will help him from time to time. Friends of the *Iceman* will occasionally drop new *Sonar Kits* in the upper-left corner of the oil field for the *Iceman* to pick up. If the *Iceman* has one or more sonar charges in his inventory, the player can press the Z key to use a charge to map the contents of the oil field within an extended radius around them, eliminating the need to dig into the ice to determine its contents. In addition to the *Sonar Kits* dropped by the *Iceman's* friends, he also starts out with one sonar charge at the start of each level.

There are two types of protesters: *Regular Protesters* and *Hardcore Protesters*. As the gameplay progresses on each level, the game will introduce various new *Regular* and *Hardcore Protesters* in the upper-right corner of the screen, who will then seek out the *Iceman* to yell at him.

Regular Protesters are not too dedicated, and are more likely to leave the oil field if they are either irritated by being squirted or bribed with gold. *Regular Protesters* wander around essentially randomly through the tunnels of the oil field. However, if a *Regular Protester* sees the *Iceman*, he will run right up to him and begin shouting at him (to irritate him). Each shout will cause the *Iceman's* health to go down, and when it reaches zero, the *Iceman* will give up (and lose a life). If a *Regular Protester* is hit by a *Boulder*, it will instantly give up and head directly for the upper-right corner of the screen, where it will disappear from the game forever (although a different, new *Protester* will likely soon replace him as the game progresses).

In contrast to the *Regular Protesters*, *Hardcore Protesters* are much nastier. They behave almost identically to *Regular Protesters*, with a few key differences. First, they're much more dedicated and so it takes more squirts from the *Iceman's* squirt gun to irritate them sufficiently to make them leave the oil field. Second, *Hardcore Protesters* cannot be bribed with *Gold Nuggets* to leave the oil field. Like *Regular Protesters*, they will stop to pick up *Nuggets* dropped by the *Iceman* if they run into them. However, after a brief delay to examine the *Nugget* (giving the *Iceman* valuable time to get away) they will pocket the gold and continue to hunt through the oil field for the *Iceman*. Third, *Hardcore Protesters* are so dedicated that they have found a way to track the *Iceman's* location from his cell phone, and if they are within a short range of the *Iceman* (e.g., less than 16 horizontal/vertical moves away – this distance increases with each level of the game), they will run toward him. Assuming they're close enough, the *HardCore Protesters* will home in on the *Iceman* and

move toward him even if they have no direct line of sight to him (Hint: use a breadth-first search like we learned about in the Queue lecture). Like *Regular Protesters*, *HardCore Protesters* are instantly super-annoyed if they're bonked by a *Boulder* and will leave the oil field immediately. And *Hardcore Protesters* will exhibit the same movement behavior as *Regular Protesters* if they're not within cell-phone-detecting range of the *Iceman*.

If the *Iceman's* health reaches zero because they are repeatedly shouted at and irritated, he gives up and loses one "life." If, after losing a life, the player has one or more remaining lives left, a new oil field is generated for them from scratch (with the same difficulty as the previous oil field they just tried to solve, but with a random, new layout), the *Iceman* is placed back on this level and they must again solve the entire level from scratch. The *Iceman* will restart the level with full health points, as well as their initial water supply for 5 squirt gun squirts, and one sonar charge.

The game window is exactly 64 squares wide by 64 squares high. Each object in the game (e.g., the *Iceman*, *Protesters*, *Boulders*, *Gold Nuggets*, *Sonar Kits*, *Barrels* of oil, *Water,* etc.) has an image that is exactly 4 squares high by 4 squares wide. The *Iceman* may move anywhere within the oil field except on top of a *Boulder*. *Protesters* are unable to dig through the *Ice* (they aren't diggers, after all) so they may only move to locations that have no *Ice* or *Boulders* (i.e., down the tunnels the Iceman has already dug). Both the *Iceman* and the *Protesters* may occupy overlapping areas of the oil field with each other.

The bottom-leftmost square of the game window has coordinates x=0,y=0, while the upper-rightmost square has coordinate x=63,y=63. Notice that x increases to the right and y increases upward toward the top of the screen. You can look in our provided file, *GameConstants.h*, for constants that represent the game window's width and height. Each object (e.g., the *Iceman* or a *Gold Nugget*) has an x,y coordinate designating its location, and this coordinate points to the lower-left-hand corner of the object. Therefore, a game object at location x=10,y=16 would occupy the 4x4 square described by a lower-left-hand corner of (10,16) and an upper-right-hand corner of (13,19). No game object may have a lower-left-hand coordinate greater than x=60 or y=60, as this would cause part of the game object to be rendered outside the oil field. Similarly, no game object may have an x or y coordinate that is less than zero.

The oil field, which is comprised of 1x1 *Ice* objects, occupies the entire width of the game window, but only the lower 60 squares of the screen. A single tunnel is always at the center of the oil field at the start of each level.

As the game progresses, each level will pose new challenges for the *Iceman*. The number of *Barrels of oil* that must be collected increases, the number of *Protesters* increases, the frequency of *Water* and *Sonar Kits* available to the *Iceman* declines, and the *Protesters* move more quickly and are stunned for shorter and shorter periods of time when squirted or when they find a *Nugget*. All of these details are described in the later sections of this document.

Once a new oil field (all of the *Ice* that comprises the 64 -square wide by 60-square high oil field) and its contents have been generated (*Boulders*, *Barrels of oil*, and *Gold Nuggets* are distributed through the field) and the *Iceman* has been added to the game, the game play begins. Game play is divided into *ticks*, and there are dozens of ticks per second (to provide smooth animation and game play). During each tick, the following occurs:

1. The player has an opportunity to hit a key to move the *Iceman* exactly one square horizontally or vertically, shoot their squirt gun (if the *Iceman* has water), use a sonar charge to reveal the nearby contents of the oil field, or give up (some levels may be unsolvable, so the player can press the Escape key to lose a life and restart the level from scratch).
2. Every other object in the game (e.g., *Regular Protesters*, *Hardcore Protesters*, *Gold Nuggets*, *Boulders*, etc.) with the exception of *Ice* objects (which don't do anything) is given an opportunity to do something. For example, when given the opportunity to do something, a *Regular Protester* can move one square (left, right, up or down) according to its built-in movement algorithm (the *Regular Protester* movement algorithms are described in detail in the various *Regular Protester* sections below).
3. One or more new game objects (e.g., *Protesters*, *Sonar Kits* or *Water*) may be added to the game; once added, they'll have an opportunity to do something during each tick as well.
4. One or more existing game objects may be removed from the game. For example, if the *Iceman* picks up a *Gold Nugget*, its object must be deleted from the game (once the *Iceman's* inventory has been updated).

During game-play, the player controls the direction of the *Iceman* with the arrow keys, or for lefties and others for whom the arrow key placement is awkward, WASD or the numeric keypad: up is *w* or *8*, left is *a* or *4*, down is *s* or *2*, right is *d* or *6*. The player may move the *Iceman* anywhere in the oil field as they please. The player can sacrifice one life and restart the current level by pressing the Escape key at any time.

The *Iceman* may shoot their squirt gun by pressing the spacebar key. If the *Iceman* manages to squirt a *Regular Protester* or a *Hardcore Protester* enough, the *Protester* will give up, and the player earns points:

For causing a *Regular Protester* to give up:	100 points
For causing a *Hardcore Protester* to give up:	250 points

In addition to squirting *Protesters*, the *Iceman* can dig the *Ice* from under a *Boulder* causing it drop on *Protesters* (of both types). If a *Boulder* bonks a *Protester* as it falls, the player earns lots of points:

For causing a *Boulder* to bonk a *Protester*:	500 points

The player can also earn points by dropping a *Gold Nugget* in front of a *Protester*. Should the *Protester* pick up a dropped *Gold Nugget*, the player earns:

For bribing a *Regular Protester* with a Gold Nugget: 25 points
For bribing a *Hardcore Protester* with a Gold Nugget: 50 points

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

- *Barrel of oil*: 1000 points (the *Iceman* must collect all of the *Barrels* on the oil field to complete the level)
- *Gold Nugget*: 10 points (the *Iceman* gets 1 piece of gold added to their inventory, which they can subsequently drop to bribe *Protesters*)
- *Water*: 100 points (5 squirts worth of water are added to the *Iceman's* inventory)
- *Sonar Kit*: 75 points (the *Iceman* gets two new sonar charges added to their inventory)

The player starts with three lives. The player loses a life if their health reaches zero (from being annoyed by *Protester* shouts, or from being bonked by a *Boulder*).

When the *Iceman* is completely annoyed (their health reaches zero), the current level ends and the player's number of remaining lives is decremented by 1. If the player still has at least one life left, then they are prompted to continue and given another chance by restarting the current level from scratch. If the player has no lives left, then the game is over. Pressing the *q* key lets you quit the game prematurely.

So how does a video game work?

Fundamentally, a video game is composed of a bunch of objects (i.e., class variables); in Iceman, those objects include the *Iceman*, *Regular Protesters*, *Hardcore Protesters*, *Goodies* (e.g., *Sonar Kits*, *Gold Nuggets*, *Water*, *Barrels of oil*) that can be picked up, *Boulders*, *Ice*, and *Squirts* (water squirted by the *Iceman* at *Protesters*). Let's call these objects "actors," since each object is an actor in our video game. Each actor has its own x, y location in the oil field, its own internal state (e.g., the location of the *Protester*, what direction the *Protester* is 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 oil field. In the case of the *Iceman*, the algorithm that controls the *Iceman's* actor object is the player's own brain and hand, and the keyboard! In the case of other actors (e.g., *Protesters*), 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 video game calls upon each object's behavioral algorithm and asks the object to perform its behavior. When asked to perform its behavior, each object's behavioral algorithm must decide what to do and then make a change to the object's state (e.g., move the object 1 square to the left), or change other objects' states (e.g., when a *Protester's* algorithm is called by the game, it may determine that it has moved next to a *Iceman*, and it may shout at the *Iceman,* irritating him). Typically the behavior exhibited by an object during a single tick is limited in order to ensure that the gameplay is smooth and that things don't move too quickly and confuse the player. For example, a *Protester* will move at most square left/right/up/down during each tick, rather than moving two or more squares; if a *Protester* were to move 5 squares in a single tick, for example, the user would be confused 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 actor's states), our game framework (that we provide) animates the actors onto the screen in their new configuration. So if a *Protester* changed its location from x=10, y=5 to x=11, y=5 (moved one square right), then our game framework would erase the graphic of the *Protester* from location 10,5 on the screen and draw the Protester's graphic at 11,5 instead. Since this process (asking actors to do something, then animating them to the screen) happens some 10-20 times per second, the user will see smooth-ish animation.

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

Assuming the ticks are quick enough (a fraction of a second), and the actions performed by the objects are subtle enough (i.e., a *Protester* 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 over time, it looks like 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 the actors (which are C++ objects) and placing them in the game world so that they will appear in the oil field.

Game play: Game play is broken down into a bunch of ticks. During each tick, all of the actors in the game have a chance to do something, and perhaps die. During each tick, new actors may be added to the game and actors who die must be removed from the game world and deleted.
Cleanup: This phase occurs either when the player has lost a life (but has more lives left), the player has completed the current level, or the player has lost all of their lives and the game is over. This phase frees all of the objects in the World (e.g., *Regular Protesters*, *Ice*, *Boulders*, *Goodies*, the *Iceman*, etc.) since the level has ended. If game-play is not over (i.e., the player has more lives), then the game proceeds back to the

Initialization step, where the oil field is repopulated with new occupants and game play restarts at the current level.

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

```
while (The player has lives left)
{
   Prompt_the_user_to_start_playing(); // "press a key to start"

   Initialize_the_game_world();  // you're going to write this func

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

         // you're going to write code to do the following
         Ask_all_actors_to_do_something();
         If_any_actors_died_then_delete_them_from_the_world();

         // we write this code to handle the animation for
         you Animate_all_of_the_alive_actors_to_the_screen();
         Sleep_for_50ms_to_give_the_user_time_to_react();
   }
   // the player died - you're going to write this code
   Cleanup_all_game_world_objects(); // you're going to write this
   if (The player has more lives)
      Prompt_the_player_to_continue();
}

Tell_the_user_the_game_is_over(); // "game over!"; we provide this
```

And here is what the `Ask_all _actors_to_do_something()` function might look like (which is one of the functions you'll have to write):

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

You will typically use a container (an array, vector, or list) to hold pointers to each of your live actors. Each actor (a C++ object) has a *doSomething( )* method. In this method, each actor (e.g., a *Regular Protester*) can decide what to do. For example, here is some pseudo code showing what a hypothetical *Regular Protester* might decide to do each time it gets asked to do something:

```
class RegularProtester: public …
{
   public:
       virtual void doSomething()
       {
            If I am facing the Iceman and he is next to me, then
                   Shout at the Iceman (to annoy him)
```

```
            Else if the Iceman is visible via direct line of sight, then
                   Switch direction to face the Iceman
                   Move one square in this direction
            Else if I'm about to run into an obstacle, then
                   Pick a new direction
                   Move one square in this direction
            Else
                   Move one square in my current direction
      }
      ...
};
```

And here's what the *Iceman's doSomething( )* method might look like:

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

What Do You Have to Do?

You must create a number of different classes to implement the Iceman game. Your classes must work properly with our provided classes, and **you must not modify our classes or our source files in any way to get your classes to work properly.** 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 (including the oil field) and all of its actors/objects (e.g., the *Iceman*, *Protesters*, *Boulders*, *Ice,* etc.) that are inside the oil field.
2. You must create a class to represent the *Iceman* in the game.
3. You must create classes for *Regular Protesters*, *Hardcore Protesters*, *Squirts* (that the Iceman shoots), *Water*, *Sonar Kits*, *Gold Nuggets*, *Ice*, *Boulders*, and *Barrels of oil*, as well as any additional base classes (e.g., a *Goodie* base class that's common to all pick-uppable items like *Water*, *Gold Nuggets*, etc., if you need one) that are required to implement the game.

You Have to Create the StudentWorld Class

Your *StudentWorld* class is responsible for orchestrating virtually all game play – it keeps track of the whole game world (the *Ice* of the oil field, and all of its inhabitants such as *Protesters*, the *Iceman*, *Boulders*, *Goodies*, etc.). It is responsible for initializing the game world at the start of the game, asking all of your game's actors to do something during each tick of the game, and destroying all of the actors in the game world when the user loses a life or when actors disappear (e.g., a *Regular Protester* leaves the oil field after being sufficiently annoyed by being repeatedly squirted).

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

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

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

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

Your *init()* method is responsible for creating the current level's oil field and populating it with *Ice*, *Boulders*, *Barrels of Oil,* and *Gold Nuggets* (we'll show you how below), and constructing a virtual representation of the current level in your *StudentWorld* class, using one or more data structures that you come up with. This function must return the value `GWSTATUS_CONTINUE_GAME` (defined in GameConstants.h).

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

Once a new level has been initialized with a call to your *init()* method, our game framework will repeatedly call your *StudentWorld's move()* method, at a rate of roughly 10-20 times per second. Each time your *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's actors (e.g., the *Iceman*, each *Regular Protester* or *Hardcore Protester*, *Boulders*, etc.) to try to do something: e.g., move themselves and/or perform their specified behavior. Finally, this method is responsible for disposing of (i.e., deleting) actors (e.g., a *Squirt* from the *Iceman's* squirt gun that has run its course, a *Regular Protester* who has left the oil field, a *Boulder* that has fallen and crashed into *Ice* below, etc.) that need to disappear during a

given tick. For example, if a *Boulder* has completed its fall and disintegrated in the *Ice below,* then its state should be set to "dead," and the after all of the actors in the game get a chance to do something during the tick, the *move()* method should remove that *Boulder* from the game world (by deleting its object and removing any reference to the object from the *StudentWorld's* data structures). Your *move()* method will automatically be called once during each tick of the game by our provided game framework. You ***will never*** call the *move()* method yourself.

The *cleanup()* method is called by our framework when the player loses a life (e.g., the *Iceman's* hit-points reach zero due to being shouted at by one or more *Protesters*), or the player successfully completes the current level. The *cleanup()* method is responsible for freeing all actors (e.g., all *Regular Protester* objects, all *Hardcore Protester* objects, all *Ice* and *Boulder* objects, the *Iceman* object, all *Goodie* objects (like *Water*, *Gold Nuggets*, *Barrels of oil*), *Squirt* objects, etc.) that are currently active in the game. This includes all actors created during either the *init()* method or introduced during subsequent game ticks (e.g., a *Hardcore Protester* that was added to the oil field during the middle of a level, or a *Squirt* of water shot by the *Iceman* just before they complete the level) that have not yet been removed from the game.

You may add as many other public or private methods and private member variables to your *StudentWorld* class as you like (in addition to the above three methods, which you *must* implement).

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

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

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

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

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

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

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

increaseScore() is used by your *StudentWorld* class (or you other classes) to increase the user's score when the *Iceman* irritates *Protesters* with a *Squirt*, picks up a *Barrel* or a *Goodie* of some sort, or bonks a *Protester* with a *Boulder,* etc. When your code calls this method, you must specify how many points the player gets (e.g., 100 points for irritating a *Regular Protester* to the point where it gives up). This means that the game score is controlled by our *GameWorld* object – you must *not* maintain your own score member variable in your own class(es).

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

Lvl: 52 Lives: 3 Hlth: 80% Wtr: 20 Gld: 3 Oil Left: 2 Sonar: 1 Scr: 321000

You'll pass in a string to this function that specifies the proper stat values.

getKey() can be used to determine if the user has hit a key on the keyboard to move the player or to fire. This method returns true if the user hit a key during the current tick, and false otherwise (if the user did not hit any key during this tick). The only argument to this method is a variable that will be filled in with the key that was pressed by the user (if any key was pressed). If the player does hit a key, the argument will be set to one of the following values (constants defined in *GameConstants.h*):

```
KEY_PRESS_LEFT
KEY_PRESS_RIGHT
KEY_PRESS_UP
KEY_PRESS_DOWN
KEY_PRESS_SPACE
KEY_PRESS_ESCAPE
KEY_PRESS_TAB
'z'
'Z'
```

The *playSound()* method can be used to play a sound effect when an important event happens during the game (e.g., a *Regular Protester* gives up due to being squirted, or the *Iceman* picks up a *Barrel of oil*). You can find constants (e.g., SOUND_PROTESTER_GIVE_UP) that describe what noise to make inside of the *GameConstants.h* file. Here's how the *playSound()* method might be used:

```
// if a Regular Protester reaches zero hit-points and dies
// then make a dying sound

if (theProtesterHasZeroHitPoints())
      GameController::getInstance().playSound(SOUND_PROTESTER_GIVE_UP);
```

init() Details

Your StudentWorld's *init()* method must:

A. Initialize the data structures used to keep track of your game's virtual world

B. Construct a new oil field that meets the requirements stated in the section below (filled with *Ice*, *Barrels of oil*, *Boulders*, *Gold Nuggets*, etc.)
C. Allocate and insert a valid *Iceman* object into the game world at the proper location

Your *init()* method must construct a representation of your virtual world and store this in your *StudentWorld* object. It is *required* that you keep track of all of the game objects (e.g., actors like *Regular Protesters*, *Gold Nuggets*, *Barrels of oil*, *Sonar Kits* , *Boulders*, etc.) ***with the exception of Ice objects and the Iceman object*** in a ***single*** STL collection like a *list* or *vector*. To do so, we recommend using a vector of pointers to your game objects, or a list of pointers to your game objects.

If you like, your *StudentWorld* class may keep a separate pointer to the *Iceman* rather than keeping a pointer to the *Iceman* object in this collection along with the other game objects.

Similarly, you may store pointers to all *Ice* objects in a different data structure than the list/vector used for your other game actors (i.e., those objects that actually do something during each tick) if you like. Hint: Keeping all of your *Ice* objects in a separate 2-D array of *Ice* pointers will speed things up.

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

Contents of Each Oil Field

First, you must completely fill rows 0 through 59 of the oil field with *Ice* objects, with the exception of a vertical mine shaft in the middle of the field. Your *Ice* class, which is used to create these *Ice* objects, must be derived in some way from our *GraphObject* class, and have an *imageID* of `IMID_ICE`. A *Ice* object is the simplest type of game object in *Iceman*. All it does is display itself to the screen – it doesn't move or perform any other actions on its own. You'll find more details on the requirements for the *Ice* object in its section below.

As mentioned above, a single tunnel, 4 squares wide (occupying columns 30-33 of the oil field), and 56 squares deep (occupying rows 4-59) must lead from the surface of the mine down into its depths, and must be devoid of any *Ice* objects.

The *Iceman* must start the game at location x=30, y=60, just atop the tunnel, at the start of each level (and after the *Iceman* loses a life and restarts a level).

You must distribute the following game objects randomly in the oil field:

B Boulders in each level, where:

int B = min(current_level_number / 2 + 2, 9)

G Gold Nuggets in each level, where:

int G = max(5-current_level_number / 2, 2)

L Barrels of oil in each level, where:

int L = min(2 + current_level_number, 21)

The starting level # is level 0, so *level 0* would have 2 *Boulders*, 5 *Nuggets* and 2 *Barrels of oil*. Or, for example, *level 2* would have 3 *Boulders*, 4 *Nuggets* and 4 *Barrels of oil*.

No distributed game object may be within a radius (Euclidian distance) of 6 squares of any other distributed game object. For example, if a *Boulder* were distributed to x=1,y=2, then a *Nugget* could not be distributed to x=6,y=4 because the two would be 5.39 squares away (less than or equal to 6 squares away). However the same *Nugget* could be distributed to x=6,y=6 because this would be 6.4 squares away (more than 6.0 squares away). *Nuggets* and *Oil Barrels* must be distributed between x=0,y=0 and x=60,y=56 inclusive, meaning that the lower-left corner of any such object must fall within this rectangle. *Boulders* must be distributed between x=0,y=20 and x=60,y=56, inclusive (so they have room to fall).

All distributed *Gold Nuggets* must start in a state that is pickup-able by the *Iceman*, but not by *Protesters*. All distributed *Gold Nuggets* must start out in a permanent state.

All distributed *Gold Nuggets* and *Barrels of oil* must start out in an invisible state (not displayed on the screen). They will become visible when the *Iceman* either gets near them (this is detailed within the specs for *Nuggets* and *Barrels*) or if the *Iceman* uses a sonar charge to scan the nearby *Ice* around him.

There must not be any *Ice* overlapping the 4x4 square region of each Boulder, so you'll need to clear this *Ice* out when you place your *Boulders* within the oil field (or place your *Boulders* first, then avoid placing *Ice* objects where the *Boulders* are located). The other items must have the area under their 4x4 image completely filled with *Ice* (in other words, these items ***must not*** be distributed at the surface of the oil field or within the mine shaft).

Once your *init()* method has distributed all of the *Ice, Iceman*, and game objects throughout the oil field, it should return so our game framework can start the game.

move() Details

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

1. It must update the status text on the top of the screen with the latest information (e.g., the user's current score, the remaining bonus score for the level, etc.).

2. It must ask all of the actors that are currently active in the game world to do something (e.g., ask a *Regular Protester* to move itself, ask a *Boulder* to see if it needs to fall down because *Ice* beneath it was dug away, give the *Iceman* a chance to move up, down, left or right, etc.).
    A. If an actor does something that causes the *Iceman* to give up, then the *move()* method should immediately return `GWSTATUS_PLAYER_DIED`.
    B. If the *Iceman* collects all of the *Barrels of oil* on the level (completing the current level), then the *move()* method should immediately play a sound of SOUND_FINISHED_LEVEL and then return a value of `GWSTATUS_FINISHED_LEVEL`.
3. It must then delete any actors that need to be removed from the game during this tick and remove them from your STL container that tracks them. This includes, for example:
    1. A *Protester* that has run to the upper-right-hand corner of the oil field after being sufficiently annoyed (by being squirted by a *Squirt* or hit by a *Boulder*) and is ready to "leave" the oil field
    2. A *Boulder* that has fallen down a shaft and disintegrated upon hitting the bottom (or another *Boulder*)
    3. A *Gold Nugget* that has been picked up by the *Iceman* or a *Protester* and is therefore no longer in the oil field
    4. A *Water Pool* that has dried up after a period of time.
    5. A *Squirt* from the *Iceman's* squirt gun once it's reached the maximum distance it can travel.
    6. Etc.

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

```
GWSTATUS_PLAYER_DIED
GWSTATUS_CONTINUE_GAME
GWSTATUS_FINISHED_LEVEL
```

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

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

The final return value indicates that the player has completed the current level (that is, gathered all of the *Barrels of oil* on the level). If your *move()* method returns this value,

then the current level is over, and our framework will call your *cleanup()* method to destroy the level, advance to the next level, then call your *init()* method to prepare that level for play, etc…

IMPORTANT NOTE: The "skeleton" code that we provide to you is hard-coded to return a `GWSTATUS_PLAYER_DIED` status value from our dummy version of the *move()* method. Unless you change this value to `GWSTATUS_CONTINUE_GAME` your game will not display anything on the screen! So if your screen just shows up black once the user starts playing, you'll know why!

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

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

   // The term "Actors" refers to all Protesters, the player, Goodies,
   // Boulders, Barrels of oil, Holes, Squirts, the Exit, etc.

    // Give each Actor a chance to do something
    for each of the actors in the game world
    {
        if (actor[i] is still active/alive)
        {
          // ask each actor to do something (e.g. move)
          tellThisActorToDoSomething(actor[i]);
          if (theplayerDiedDuringThisTick() == true)
                return GWSTATUS_PLAYER_DIED;
          if (theplayerCompletedTheCurrentLevel() == true)
          {
                return GWSTATUS_FINISHED_LEVEL;
          }
        }
    }

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

    // return the proper result
    if (theplayerDiedDuringThisTick() == true)
          return GWSTATUS_PLAYER_DIED;
    // If the player has collected all of the Barrels on the level, then
    // return the result that the player finished the level
    if (theplayerCompletedTheCurrentLevel() == true)
    {
          playFinishedLevelSound();
          return GWSTATUS_FINISHED_LEVEL;
    }

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

Give Each Actor a Chance to Do Something

During each tick of the game each active actor must have an opportunity to do something (e.g., move around, shoot, etc.). Actors include the *Iceman*, *Regular Protesters*, *Hardcore Protesters*, *Boulders*, *Gold Nuggets*, *Barrels of oil*, *Water*, *Squirts* from the *Iceman's* squirt gun, and *Sonar Kits*.

Your *move()* method must enumerate each active actor in the oil field (i.e., held by your *StudentWorld* object) and ask it to do something by calling a method in the actor's object named *doSomething()*. In each actor's *doSomething()* method, the object will have a chance to perform some activity based on the nature of the actor and its current state: e.g., a *Regular Protester* might move one step up, the *Iceman* might shoot a *Squirt* of water, a *Boulder* may fall down one square, etc.

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.

Add New Actors During Each Tick

During each tick of the game in your *move()* method, you may need to add new *Protesters* (*Regular* or *Hardcore*) and/or *Goodies* (*Water Pools* or *Sonar Kits*) to the oil field. You must use the following approach to decide whether to add these new actors to the oil field:

1. A new *Protester* (*Regular* or *Hardcore*) may only be added to the oil field after at least T ticks have passed since the last *Protester* of any type was added, where:

   int T = max(25, 200 – current_level_number)

2. The target number P of *Protesters* that should be on the oil field is equal to:

   *int P = min(15, 2 + current_level_number * 1.5)*

   However, based on #1 above, you can only add a new *Protester* to the oil field every T ticks, so the actual number of *Protesters* on the oil field at any particular time may be less than the target number P.

3. The first *Protester* must be added to the oil field during the very first tick of each level (and any replays of the level).
4. Assuming the appropriate number of ticks T has elapsed since the last *Protester* was added to the oil field, AND the current number of *Protesters* on the oil field is less than P, then you must add a new *Protester* to the oil field during the current

tick. All *Protesters* must start at location x=60,y=60 on the screen. The odds of the *Protester* being a *Hard Core Protester* (vs. a *Regular Protester*) must be determined with this formula:

*int probabilityOfHardcore = min(90, current_level_number * 10 + 30)*

5. There is a 1 in G chance that a new *Water Pool* or *Sonar Kit* Goodie will be added to the oil field during any particular tick, where:

   *int G = current_level_number * 25 + 300*

   Assuming a new Goodie should be added, there is a 1/5 chance that you should add a new *Sonar Kit*, and a 4/5 chance you should add a *Water* Goodie.

   Each new *Sonar Kit* must be added at x=0, y=60 on the screen.

   Each new *Water* Goodie must be added to a random ice-less spot in the oil field. *Water* may only be added to a location if the entire 4x4 grid at that location is free of *Ice*.

Remove Dead Actors after Each Tick

At the end of each tick your *move()* method must determine which of your actors are no longer alive, remove them from your STL container of active actors, and delete their objects (so you don't have a memory leak). For example, once a *Barrel* is picked up by the *Iceman* during a tick, it should be marked as "not active." After giving all of the actors a chance to move during the current tick, your *move()* method would then discover this inactive *Barrel* (as well as any other objects that have become inactive during this tick) and remove its object *pointer* from your *StudentWorld's* container of active objects. Finally, your *move()* method should delete the object (using the C++ *delete* command) to free up room in memory for future actors that will be introduced later in the game. (Hint: Each of your actors could have a member variable indicating whether or not it is still active/alive!)

Updating the Display Text

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

```
void setDisplayText()
{
   int level = getCurrentGameLevel();
   int lives = getNumLivesLeft();
   int health = getCurrentHealth();
   int squirts = getSquirtsLeftInSquirtGun();
   int gold = getPlayerGoldCount();
   int barrelsLeft = getNumberOfBarrelsRemainingToBePickedUp();
```

```
    int sonar = getPlayerSonarChargeCount();
    int score = getCurrentScore();

    // Next, create a string from your statistics, of the form:
    //   Lvl: 52 Lives: 3 Hlth: 80% Wtr: 20 Gld: 3 Oil Left: 2 Sonar: 1 Scr: 321000


    string s = someFunctionYouUseToFormatThingsNicely(level, lives, health,
                                squirts, gold, barrelsLeft, sonar, score);

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

Your status line must meet the following requirements:

1. Each field must be exactly as wide as shown in the example above:
    a. The Lvl field must be 2 digits long, with leading spaces (e.g., "_1", where _ represents a space).
    b. The Lives field should be 1 digit long (e.g., "2").
    c. The Hlth field should be 3 digits long and display the player's health percentage (not its hit-points!), with leading spaces, and be followed by a percent sign (e.g., "_70%").
    d. The Wtr field should be 2 digits long, with a leading space as required (e.g., "_ 7").
    e. The Gld field should be 2 digits long, with a leading space as required (e.g., "_ 3").
    f. The Oil Left field should be 2 digits long, with a leading space as required (e.g., "_ 4").
    g. The Sonar field should be 2 digits long, with a leading space as required (e.g., "_ 2").
    h. The Scr must be exactly 6 digits long, with leading zeros (e.g., 003124).
2. Each statistic must be separated from the last statistic by two spaces. For example, between the "000100" of the score and the "L" in "Level" there must be exactly two spaces.

cleanUp() Details

When your *cleanUp()* method is called by our game framework, it means that the *Iceman* lost a life (e.g., his hit-points/annoyance tolerance reached zero due to being shouted at or being bonked by a *Boulder*), or has completed the current level. In this case, every actor in the entire oil field (the *Iceman* and every *Protester*, *Goodies* like *Nuggets*, *Sonar Kits* and *Water*, *Barrels of oil*, *Boulders*, *Ice*, etc.) must be deleted and removed from your *StudentWorld's* container(s) of active objects, resulting in an empty oil field. If the player has more lives left, our provided code will subsequently call your *init()* method to create a new oil field and the level will then continue from scratch with a brand new set of actors (including a newly-generated *Iceman*!).

You ***must not*** call the *cleanUp()* method yourself when the player's hit points go to zero. Instead, this method will be called by our code

You Have to Create Classes for All Actors

The Iceman game has a number of different Game Objects, including:

- The *Iceman*
- *Regular Protesters*
- *Hardcore Protesters*
- *Squirts of water (that can be shot by the Iceman)*
- *Barrels of oil*
- *Boulders*
- *Gold Nuggets*
- *Sonar kits*
- *Water* refills (that can be picked up to refill the Iceman's squirt gun)
- *Ice*

Each of these game objects can occupy a slot in the oil field and interact with other game objects within the oil field.

Now of course, many of your game objects will share things in common – for instance, every one of the objects in the game (*Regular Protesters*, the *Iceman*, *Ice*, *Boulders*, etc.) has an x,y coordinate. Many game objects have the ability to perform an action – to do something (e.g., move or shoot) – during each tick of the game. Many of them can be potentially annoyed (e.g., the *Iceman* and *Protesters* can be by annoyed by *Squirts* of water or bonked by *Boulders*, etc.) and could be irritated to the point of quitting during a tick. All of them need some attribute that indicates whether or not they are still active/alive (or they became inactive/died during the current tick), etc.

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

Your grade on this project will largely depend upon your ability to create an intelligent set of classes that follow good object-oriented design principles. Your classes MUST NEVER duplicate code or member variables – if you find yourself writing the same (or largely similar) code across multiple classes, then this is an indication that you should define a common base class and migrate this common functionality/data to the base class. Duplication of code is the root of all evil in programming – it leads to bugs, inconsistencies, code bloat, etc. and should be avoided whenever possible.

Hint: When you notice this specification repeating the same text nearly exactly in the following sections (e.g., in the *Regular Protester* section and the *Hardcore Protester*

section, or in the *Gold Nugget* or *Sonar Kit* sections) you MUST make sure to identify common behaviors and move these into proper base classes. NEVER duplicate behaviors across classes that can be moved into a base class!

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

```
class Actor: public GraphObject {
public:
        …
};

class Protester: public Actor {
public:
        …
};

class HardcoreProtester: public Protester {
public:
        …
};
```

GraphObject is a class that we have defined that helps hide a lot of 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 in your classes:

```
GraphObject(int imageID, int startX, int startY,
        DIRECTION startDirection, float size = 1.0,
        unsigned int depth = 0);
void setVisible(bool shouldIDisplay);
void getX() const;
void getY() const;
void moveTo(int x, int y);
DIRECTION getDirection() const;    // Directions: up, down, left, right
void setDirection(DIRECTION d);    // Directions: up, down, left, right
```

You may use any of these methods in your derived classes, but you ***must not*** use any other methods found inside of *GraphObject* in your other classes (even if they are public in our class). You ***must not*** redefine any of these methods in your derived classes since they are not defined as virtual in our base class.

GraphObject(
int imageID,
int startX,
int startY,
DIRECTION startDirection,
float size = 1.0,

unsigned int depth = 0

)

When you construct a new *GraphObject*, you must specify the following parameters:

1. An *imageID* that indicates what graphical image (aka sprite) our graphics engine should display on the screen. One of the following IDs, found in *GameConstants.h*, MUST be passed in for the *imageID* value:

```
IMID_PLAYER                 // for the Iceman
IMID_PROTESTER              // a regular protester
IMID_HARD_CORE_PROTESTER    // a hardcore protester
IMID_WATER_SPURT            // for a squirt of water from the Iceman
IMID_BOULDER
IMID_BARREL                 // a barrel of oil
IMID_ICE                    // a 1x1 square of ice
IMID_GOLD                   // a gold nugget
IMID_SONAR                  // a sonar kit
IMID_WATER_POOL             // a water pool to refill the squirt gun
```

2. You must specify the initial *x*,*y* location of the object. The *x* value may range from 0 to VIEW_WIDTH-SPRITE_WIDTH inclusive, and the *y* value may range from 0 to VIEW_HEIGHT-SPRITE_HEIGHT inclusive. Notice that you pass the coordinates as *x*,*y* (i.e., column, row and ***not*** row, column.).

3. You must specify the initial direction that each game object will be facing. Acceptable values include: *up*, *down*, *left*, and *right*. The definitions for these values may be found in the *GraphObject* class within *GraphObject.h.*

4. You must specify the size of the image. Most images (e.g., *Protesters*, *Gold Nuggets*, the *Iceman*, etc.) will have a size of 1.0, which is the equivalent of a 4 squares by 4 squares on the screen. In contrast, *Ice* will always have a size of .25 making it 1 square by 1 square.

5. You must specify the depth of the image. This value indicates whether the image is in the foreground (0), in the background (10) or somewhere in between. If two images are displayed at the same location, the graph object with the smaller depth value will be drawn on top of the graph object with the larger depth value, either partially or completely obscuring it. We will specify what the depth should be for each of your *GraphObject*-derived classes in the sections below.

setVisible(bool shouldIDisplay)

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

setVisible(bool shouldIDisplay) is used to tell our graphical system whether or not to display a particular *GraphObject* on the screen.

If you call *setVisible(true)* on a *GraphObject*, then your object will be displayed on screen automatically by our framework (e.g., a *Regular Protester* image will be drawn to the screen at the *GraphObject's* specified x,y coordinates if the object's imageID is IMID_PROTESTER). If you call *setVisible(false)* then your GraphObject will not be displayed on the screen.

NOTE: When you create a new game object, always remember that if you want it to be displayed on the screen, you must call its *setVisible()* method with a value of *true*!

int getX() const
int getY() const

These methods are used to determine a *GraphObject's* current location on the screen. Since each *GraphObject* maintains an x,y location, this means that your derived classes ***must not*** also have x,y member variables, but instead use those in the *GraphObject* base class.

void moveTo(int x, int y)

The *moveTo()* method ***must*** be used to update the location of a *GraphObject* within the oil field. For example, if a *Regular Protester's* movement logic dictates that it should move to the right, you must do something like the following:

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

You **must** use the *moveTo()* method to adjust the location of a *GameObject* in the game 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).

DIRECTION getDirection() const
void setDirection(DIRECTION d)

These two methods are used to get and set the current direction the *GraphObject* is facing. Valid values include *up*, *down*, *left*, and *right*. Changing the direction causes our graphics engine to automatically rotate the image appropriately before displaying it on the screen.

Now that you know about our *GraphObject* base class, you'll need to derive all of your game object classes based on this class (either directly, or indirectly).

The Iceman Class

Here are the requirements you must meet when implementing the *Iceman* class:

What a *Iceman* Object Must Do When It Is Created

When it is first created:

1. The *Iceman* must have an image ID of IMID_PLAYER.
2. The *Iceman* must always start at location x=30, y=60.
3. The *Iceman*, in its default state:
    a. Is given 10 hit points
    b. Is given 5 units of water to squirt with his squirt gun (he may pick up additional *Water* in the oil field)
    c. Is given 1 sonar charge
    d. Starts out with zero gold nuggets
    e. Should start facing rightward
4. The *Iceman* has the following graphic parameters:
    a. It has an image depth of 0 – meaning its graphic image should always be in the foreground (above other images)
    b. It has a size of 1.0

In addition to any other initialization that you decide to do in your *Iceman* class, a *Iceman* (or one of its base classes) must set itself to be visible using the *GraphObject* class's *setVisible()* method, e.g.:

```
setVisible(true);
```

What the *Iceman* Must Do During a Tick

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

1. The *Iceman* must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. If the *Iceman's* 4x4 image currently overlaps any *Ice* objects within the oil field, then it will call upon the *StudentWorld* object to:
    a. Remove/destroy the *Ice* objects from the 4x4 area occupied by the *Iceman* (from x, y to x+3,y+3 inclusive)
    b. Make a digging sound by playing the SOUND_DIG sound (see the *StudentWorld* section of this document for details on how to play a sound).

3. Otherwise, the *doSomething()* method must check to see if the human player pressed a key (the section below shows how to check if the user pressed a key, and if so, how to determine what key the user pressed).
    a. If the user presses the Escape key, this allows the user to abort the current level. In this case, the *Iceman* object should set itself to completely annoyed (aka dead). The code in your *StudentWorld* class should detect that the *Iceman* has died and address this appropriately (e.g., replay the level from scratch, or end the game if the player has run out of lives).
    b. If the user presses the spacebar key, then the *Iceman* will fire a *Squirt* into the oil field, assuming they have sufficient water in their squirt gun to do so. The *Iceman* will then reduce their water count by 1. To fire a *Squirt*, the player must create and add a new *Squirt* object into the oil field at a location that is four squares immediately in front of the *Iceman*, facing the same direction as the *Iceman*.

        So if the *Iceman* is at x=10,y=7, and he is facing upward, then the new *Squirt* object would be placed at location x=10, y=11, and the *Squirt* would be facing upward.

        Every time the *Iceman* fires a *Squirt*, it must play the sound SOUND_PLAYER_SQUIRT.

        If the initial location where the *Squirt* starts (i.e., 4 squares in front of the *Iceman*) is occupied by one or more *Ice* objects or is within a radius of 3.0 of a *Boulder* object, then the *Iceman* must still make a squirting noise and will waste 1 unit of water, but the *Iceman* must not add any *Squirt* object into the oil field and thus no damage can be done to any *Protesters* (it's as if the *Iceman* shot water against a wall and it immediately dissipated).

        Hint: One way to implement this is to have your *Iceman* create a new *Squirt* object in the proper location and facing the proper direction, and then give it to your *StudentWorld* to manage (i.e., to animate) along with your other game objects.

    c. If the player wants their *Iceman* to turn to face a different direction then it is currently facing (by pressing one of the arrow keys), then the *Iceman's* direction should be adjusted to the indicated direction *but the Iceman must not move in that direction*. For example, if the *Iceman* is currently facing up, and the player hits the right arrow key, then the *Iceman* should turn to face right, but not move right.

    d. Else, if the *Iceman* is already facing in a particular direction and the user hits the arrow key to move in that same direction, AND the *Iceman* is allowed to move in that direction (e.g., there is no *Boulder* in the way, and the position doesn't take the *Iceman* outside of the oil field), then your code must use the

GraphObject's moveTo() method to adjust the *Iceman's* location by one square in the indicated direction.

Valid coordinates for the *Iceman* are from x=0,y=0 to x=60,y=60, inclusive. Moreover, the *Iceman* cannot occupy a square that is less than or equal to a radius of 3 away from the center of any *Boulder*.

e. If the player presses the 'Z' or 'z' keys and the *Iceman* has one or more sonar charges remaining in his inventory, then this will cause the *Iceman* to use his sonar charge to illuminate the contents of the oil field within a radius of 12 of his location:
    i. The count of sonar charges held by the *Iceman* must be decremented by 1.
    ii. All hidden game objects (e.g., *Gold Nuggets* or *Barrels* of oil) that are within a radius of 12 (e.g., this includes 11.99 squares away) must be made visible via *setVisible()* and revealed to the player.
f. If the player presses the TAB key and they have one or more units of gold in their inventory, then they must add a *Gold Nugget* object into the oil field at their current x,y location and reduce their gold count by 1 unit. The dropped *Gold Nugget* must have a temporary state and a lifetime of 100 game ticks, and is only pickup-able by *Protesters* (either *Regular* or *Hardcore*). Such a dropped *Nugget* will start out in a visible state.

What the *Iceman* Must Do When It Is Annoyed

When the *Iceman* is annoyed (i.e., shouted at by a *Protester*), its hit-points should be decremented by the appropriate amount (e.g., 2 points). Hint: All objects that can be annoyed should have some type of method that can be called to annoy them.

If the *Iceman's* hit-points reach zero or below due to being annoyed, the *Iceman* must set its object's state to dead, and then play an "I give up" sound effect: SOUND_PLAYER_GIVE_UP. Note: The *StudentWorld* class should check the *Iceman's* status during each tick and if he transitions to a dead state, it should return the appropriate value indicating that the player lost a life.

Getting Input From the User

Since Iceman is a *real-time* game, you can't use the typical *getline* or *cin* approach to get a user's key press within the player's *doSomething()* method. This would stop your program and wait until the user types in the proper data and hits the Enter key. This would make for a really boring game (requiring the user to hit a directional key then hit Enter, then hit a direction key, then hit Enter, etc.). Instead, you will need to use a special function that we provide in our *GameWorld* class (which your StudentWorld class is derived from) called `getKey()` to get input from the user[1]. This function rapidly checks

[1] Hint: Since your *Iceman* class will need to access the *getKey()* method in the *GameWorld* class (which is the base class for your *StudentWorld* class), your *Iceman* class (or more correctly, one of its base classes) will need to obtain a pointer to the *StudentWorld* class. If you look at our code example, you'll see how the

to see if the user hit a key. If the user hit a key, the function returns true and the int variable passed to it is set to the code for the key. Otherwise, the function immediately returns false, meaning that no key was hit. This function could be used as follows:

```
void Iceman::doSomething()
{
    ...
    int ch;
    if (getWorld()->getKey(ch) == true)
    {
          // user hit a key this tick!
        switch (ch)
        {
          case KEY_PRESS_LEFT:
            ... move player to the left ...;
            break;
          case KEY_PRESS_RIGHT:
            ... move player to the right ...;
            break;
          case KEY_PRESS_SPACE:
            ... add a Squirt in front of the player...;
            break;
          // etc…
        }
    }
    ...
}
```

Ice

Here are the requirements you must meet when implementing the *Ice* class.

Ice doesn't really do much. It just sits still in place.

What Ice Must Do When It Is Created

When it is first created:

1. A *Ice* object must have an image ID of IMID_ICE.
2. Each *Ice* object must have its x,y location specified for it – the *StudentWorld* class can pass in this x,y location when constructing a new *Ice* object (e.g., when constructing the entire oil field).
3. Each *Ice* object must start out facing rightward.
4. *Ice* has the following graphic parameters:
    a. It has an image depth of 3 – meaning its graphic image should always be in the background (all other game objects have smaller depth values)
    b. It has a size of .25, meaning it occupies only a 1x1 square in the oil field.

Iceman's *doSomething()* method first gets a pointer to its world via a call to *getWorld()* (a method in one of its base classes that returns a pointer to a *StudentWorld*), and then uses this pointer to call the *getKey()* method.

In addition to any other initialization that you decide to do in your *Ice* class, a *Ice* object must set itself to be visible using the *GraphObject* class's *setVisible()* method, e.g.:

```
setVisible(true);
```

What a Ice Object Must Do During a Tick

It's *ice* – what do you expect it to do? It does nothing! As such, it doesn't need to have a *doSomething()* method… Or if it does have one, it doesn't need to do anything.

What a Ice Object Must Do When It Is Annoyed

Ice objects cannot be annoyed (i.e., when a *Squirt* collides with them). After all, if you were *Ice*, would you be annoyed if you were squirted with water?

Boulders

You must create a class to represent a *Boulder*. *Boulders* basically sit around until the *Ice* underneath them has been dug away, and then after a short delay, they fall down until they hit the bottom of the shaft and disintegrate. Here are the requirements you must meet when implementing the *Boulder* class.

What a Boulder object Must Do When It Is Created

When it is first created:

1. The *Boulder* object must have an image ID of IMID_BOULDER.
2. A *Boulder* object must have its x,y location specified for it – the *StudentWorld* class can pass in this x,y location when constructing a new *Boulder* object.
3. Boulders start out in a *stable* state (more on this below).
4. Boulders start out facing downward.
5. Boulders have the following graphic parameters:
    a. They have an image depth of 1 – behind actors like *Protesters*, but above *Ice*
    b. They have a size of 1.0

In addition to any other initialization that you decide to do in your *Boulder* class, a *Boulder* object must set itself to be visible using the *GraphObject* class's *setVisible()* method, e.g.:

```
setVisible(true);
```

What the Boulder Object Must Do During a Tick

Each time the *Boulder* object is asked to do something (during a tick), it should:

1. Check to see if it's still alive. If not, its *doSomething()* method should immediately return.
2. If the *Boulder* is currently in the *stable* state, then it must check to see if there is any *Ice* in the 4 squares immediately below it. For example, if the *Boulder* is at location x=20,y=30, it would check squares (20,29), (21,29), (22,29) and (23,29) for *Ice*. If there is **any** *Ice* below the *Boulder,* it does nothing – after all it's a dumb rock. However, if none of the 4 squares beneath the *Boulder* have any *Ice*, then the *Boulder* must transition into a *waiting* state:
    A. It must enter a *waiting* state for the next 30 ticks
3. If the *Boulder* is in a *waiting* state and 30 ticks have elapsed, then it must transition into a *falling* state and play the sound SOUND_FALLING_ROCK.
4. If the *Boulder* is in a *falling* state, then:
    A. It must continue to move downward one square during each tick until it either (a) hits the bottom of the oil field (i.e., it tries to move to y=-1), (b) runs into the top of another *Boulder*, or (c) it runs into *Ice* (i.e., by moving down a square, the *Boulder* would overlap over one or more *Ice* objects). When any of the above conditions are met the *Boulder* must set its state to dead so it can be removed from the game at the end of the current tick.
    B. If the *Boulder* comes within a radius of 3 (i.e. less than or *equal* to 3 squares, so 2.99 would count) of any *Protester(s)* or the *Iceman* while falling, it must cause 100 points of annoyance to those actors (effectively completely annoying them in one fell swoop). The *Boulder* will continue falling down as described above, even if it annoys one or more *Protesters* on its way down (i.e., it won't stop falling if it runs into a *Protester*). However, if the *Boulder* annoys the *Iceman*, the player will instantly lose a life.

What a Boulder Must Do When It Is Annoyed

Boulders cannot be annoyed!

Squirt

You must create a class to represent a *Squirt* from a squirt gun. A *Squirt* object will be introduced into the oil field by the *Iceman* when he shoots his squirt gun, and will then travel 4 squares in its initially-facing direction or until it hits any *Ice*, a *Protester* of any type, or a *Boulder*. Here are the requirements you must meet when implementing the *Squirt* class.

What a Squirt object Must Do When It Is Created

When it is first created:

1. The *Squirt* object must have an image ID of IMID_WATER_SPURT.
2. All *Squirt* objects must have their x,y location specified for them by the

Iceman.
3. All *Squirt* objects must have their direction specified for them by the *Iceman*.
4. All *Squirt* objects start off with an initial travel distance of 4 squares
5. All *Squirt* objects have the following graphic parameters:
    a. They have an image depth of 1 – behind actors like *Protesters*, but above *Ice*
    b. They have a size of 1.0

In addition to any other initialization that you decide to do in your *Squirt* class, a *Squirt* object must set itself to be visible using the *GraphObject* class's *setVisible()* method, e.g.:

```
setVisible(true);
```

What the Squirt Object Must Do During a Tick

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

1. If a *Squirt* is within a radius of 3.0 of one or more *Protesters* (up to and including a distance of 3.0 squares away), it will cause 2 points of annoyance to these *Protester(s)*, and then immediately set its state to dead, so it can be removed from the oil field at the end of the tick.
2. If a *Squirt* has traveled through its full travel distance, then it immediately sets its state to dead, so it can be removed from the oil field at the end of the current tick.
3. Otherwise, the *Squirt* must check to see if it can move one square in its currently-facing direction. If this target location is occupied by either *Ice* or a *Boulder*, then the *Squirt* immediately sets its state to dead so it can be removed from the oil field at the end of the current tick.
4. Otherwise, the *Squirt* moves one square forward in its currently-facing direction, and then returns.

What a Squirt Must Do When It Is Annoyed

Squirts can't be annoyed, silly.

Barrel of Oil

You must create a class to represent a *Barrel* of oil that needs to be picked up by the *Iceman*. Here are the requirements you must meet when implementing the *Barrel* class.

What a Barrel object Must Do When It Is Created

When it is first created:

1. All *Barrels* must have an image ID of IMID_BARREL.

2. All *Barrels* must have their x,y location specified for them when they are created.
3. All *Barrels* must start off facing rightward.
4. All *Barrels* start out invisible – they must be discovered (by the *Iceman* walking in close proximity to them) before they become visible.
5. *Barrels* have the following graphic parameters:
    a. They have an image depth of 2 – behind actors like *Protesters*, but above *Ice*
    b. They have a size of 1.0

What the Barrel Object Must Do During a Tick

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

1. The object must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if the *Barrel* is not currently visible AND the *Iceman* is within a radius of 4.0 of it (<= 4.00 units away), then:
    a. The *Barrel* must make itself visible with the *setVisible()* method.
    b. The *Barrel's doSomething()* method must immediately return.
2. Otherwise, if the *Barrel* is within a radius of 3.0 (<= 3.00 units away) from the *Iceman*, then the *Barrel* will activate, and:
    a. The *Barrel* must set its state to dead (so that it will be removed by your *StudentWorld* class from the game at the end of the current tick).
    b. The Barrel must play a sound effect to indicate that the player picked up the Goodie: `SOUND_FOUND_OIL`.
    c. The Barrel must increase the player's score by 1000 points.
    d. If necessary, the *Barrel* may inform the *StudentWorld* object that it was picked up. Once all *Barrels* on the level have been picked up, the player finishes the level and may advance to the next level.

What an Barrel Must Do When It Is Annoyed

Barrels can't be attacked and will not block *Squirts* from the *Iceman's* squirt gun (they pass right over the *Barrels)*.

Gold Nugget

You must create a class to represent a *Gold Nugget.* If the *Iceman* picks up a *Gold Nugget*, he can then drop it into the oil field at a later time to bribe a *Protester* (of either type). Here are the requirements you must meet when implementing the *Gold Nugget* class.

What a Gold Nugget object Must Do When It Is Created

When it is first created:

1. All *Gold Nuggets* must have an image ID of IMID_GOLD.
2. All *Gold Nuggets* must have their x,y location specified for them when they are created.
3. All *Gold Nuggets* must start off facing rightward.
4. A *Gold Nugget* may either start out invisible or visible – this must be specified by the code that creates the *Nugget,* depending on the context of its creation. *Nuggets* buried within the *Ice* of the oil field always start out invisible, whereas *Nuggets* dropped by the *Iceman* start out visible.
5. A *Gold Nugget* will either be pickup-able by the *Iceman* or pickup-able by *Protesters*, but not both. This state must be specified by the code that creates the *Gold Nugget* object.
6. A *Gold Nugget* will either start out in a *permanent* state (where they will remain in the oil field until they are picked up by the *Iceman* or the level ends) or a *temporary* state (where they will only remain in the oil field for a limited number of ticks before disappearing or being picked up by a *Protester*). This state must be specified by the code that creates the *Gold Nugget* object.
7. *Gold Nuggets* have the following graphic parameters:
    a. They have an image depth of 2 – behind actors like *Protesters*, but above *Ice*
    b. They have a size of 1.0

What the *Gold Nugget* Object Must Do During a Tick

Each time the *Gold Nugget* object is asked to do something (during a tick):

1. The object must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if the *Gold Nugget* is not currently visible AND the *Iceman* is within a radius of 4.0 of it (<= 4.00 units away), then:
    e. The *Gold Nugget* must make itself visible with the *setVisible()* method.
    f. The *Gold Nugget doSomething()* method must immediately return.
3. Otherwise, if the *Gold Nugget* is pickup-able by the *Iceman* and it is within a radius of 3.0 (<= 3.00 units away) from the *Iceman*, then the *Gold Nugget* will activate, and:
    a. The *Gold Nugget* must set its state to dead (so that it will be removed by your *StudentWorld* class from the game at the end of the current tick).
    b. The *Gold Nugget* must play a sound effect to indicate that the *Iceman* picked up the Goodie: `SOUND_GOT_GOODIE`.
    c. The *Gold Nugget* increases the player's score by 10 points (This increase can be performed by the *Iceman* class or the *Gold Nugget* class).
    d. The *Gold Nugget* must tell the *Iceman* object that it just received a new *Nugget* so it can update its inventory.
4. Otherwise, if the *Gold Nugget* is pickup-able by *Protesters* and it is within a radius of 3.0 (<= 3.00 units away) from a *Protester*, then the *Gold Nugget* will activate, and:

a. The *Gold Nugget* must set its state to dead (so that it will be removed by your *StudentWorld* class from the game at the end of the current tick).
b. The *Gold Nugget* must play a sound effect to indicate that the *Iceman* picked it up: `SOUND_PROTESTER_FOUND_GOLD`.
c. The *Gold Nugget* must tell the *Protester* object that it just received a new *Nugget* so it can react appropriately (e.g., be bribed).
d. The *Gold Nugget* increases the player's score by 25 points (This increase can be performed by the *Protester* class or the *Gold Nugget* class).

Note: A *Gold Nugget* can only bribe a single *Protester* (either *Regular* or *Hardcore*) before disappearing from the game. If multiple *Protesters* are within the activating radius of the *Nugget*, then only one of the *Protesters* must be bribed.

5. If the *Gold Nugget* has a temporary state, then it will check to see if its tick lifetime has elapsed, and if so it must set its state to dead (so that it will be removed by your *StudentWorld* class from the game at the end of the current tick).

What a Gold Nugget Must Do When It Is Annoyed

Gold Nuggets can't be attacked and will not block *Squirts* from the *Iceman's* squirt gun (they pass right over the *Nuggets)*.

Sonar Kit

You must create a class to represent a *Sonar Kit.* If the *Iceman* picks up a *Sonar Kit*, he can use it to scan the oil field at a later time to locate buried *Gold Nuggets and Barrels of oil*. Here are the requirements you must meet when implementing the *Sonar Kit* class.

What a Sonar Kit object Must Do When It Is Created

When it is first created:

1. All *Sonar Kits* must have an image ID of IMID_SONAR.
2. All *Sonar Kits* must have their x,y location specified for them when they are created.
3. All *Sonar Kits* must start off facing rightward.
4. All *Sonar Kits* starts out visible.
5. A *Sonar Kit* is only pickup-able by the *Iceman*.
6. A *Sonar Kit* will always start out in a *temporary* state (where they will only remain in the oil field for a limited number of ticks before disappearing) – the number of ticks T a *Sonar Kit* will exist can be determined from the following formula:

   T = max(100, 300 – 10*current_level_number)

7. *Sonar Kits* have the following graphic parameters:
    a. They have an image depth of 2 – behind actors like *Protesters*, but above *Ice*
    b. They have a size of 1.0

In addition to any other initialization that you decide to do in your *Sonar Kit* class, a *Sonar Kit* object must set itself to be visible using the *GraphObject* class's *setVisible()* method, e.g.:

```
setVisible(true);
```

What the *Sonar Kit* Object Must Do During a Tick

Each time the *Sonar Kit* object is asked to do something (during a tick):

1. The object must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if the *Sonar Kit* is within a radius of 3.0 (<= 3.00 units away) from the *Iceman*, then the *Sonar Kit* will activate, and:
    a. The *Sonar Kit* must set its state to dead (so that it will be removed by your *StudentWorld* class from the game at the end of the current tick).
    b. The *Sonar Kit* must play a sound effect to indicate that the *Iceman* picked up the Goodie: `SOUND_GOT_GOODIE`.
    c. The *Sonar Kit* must tell the *Iceman* object that it just received a new *Sonar Kit* so it can update its inventory.
    d. The *Sonar Kit* increases the player's score by 75 points (This increase can be performed by the *Iceman* class or the *Sonar Kit* class).
3. Since the *Sonar Kit* is always in a temporary state, it will check to see if its tick lifetime has elapsed, and if so it must set its state to dead (so that it will be removed by your *StudentWorld* class from the game at the end of the current tick).

What an Sonar Kit Must Do When It Is Annoyed

Sonar Kits can't be annoyed and will not block *Squirts* from the *Iceman's* squirt gun.

Water Pool

You must create a class to represent a *Water Pool*. If the *Iceman* picks up *Water*, he can use it to increase the contents of his squirt gun. Here are the requirements you must meet when implementing the *Water Pool* class.

What a Water Pool object Must Do When It Is Created

When it is first created:

1. All *Water Pools* must have an image ID of IMID_WATER_POOL.
2. All *Water Pools* must have their x,y location specified for them when they are created.
3. All *Water Pools* must start off facing rightward.
4. All *Water Pools* start out visible.
5. A *Water Pool* is only pickup-able by the *Iceman*.
6. A *Water Pool* will always start out in a *temporary* state (where they will only remain in the oil field for a limited number of ticks before disappearing) – the number of ticks T that a *Water Pool* will exist can be determined from the following formula:

   T = max(100, 300 – 10*current_level_number)

7. *Water Pools* have the following graphic parameters:
   a. They have an image depth of 2 – behind actors like *Protesters*, but above *Ice*
   b. They have a size of 1.0

In addition to any other initialization that you decide to do in your *Water Pool* class, a *Water Pool* object must set itself to be visible using the *GraphObject* class's *setVisible()* method, e.g.:

```
setVisible(true);
```

What a *Water* Pool Object Must Do During a Tick

Each time a *Water* pool object is asked to do something (during a tick):

1. The object must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if the *Water* pool is within a radius of 3.0 (<= 3.00 units away) from the *Iceman*, then the *Water* pool will activate, and:
   e. The *Water* pool must set its state to dead (so that it will be removed by your *StudentWorld* class from the game at the end of the current tick).
   f. The *Water* pool must play a sound effect to indicate that the *Iceman* picked up the Goodie: `SOUND_GOT_GOODIE`.
   g. The *Water* pool must tell the *Iceman* object that it just received 5 new squirts of water so it can update its inventory.
   h. The *Water* pool increases the player's score by 100 points (This increase can be performed by the *Iceman* class or the *Sonar Kit* class).

3. Since the *Pool of Water* is always in a temporary state, it will check to see if its tick lifetime has elapsed, and if so it must set its state to dead (so that it will be removed by your *StudentWorld* class from the game at the end of the current tick).

What a Water Pool Must Do When It Is Annoyed

Water Pools can't be annoyed and will not block *Squirts* from the *Iceman's* squirt gun.

Regular Protesters

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

What a Regular Protester object Must Do When It Is Created

When it is first created:

1. Each *Regular Protester* must have an image ID of IMID_PROTESTER.
2. Each *Regular Protester* must start out facing left.
3. Each *Regular Protester* must decide how many squares, *numSquaresToMoveInCurrentDirection*, it will decide to move left before possibly switching its direction. This value must be: 8 <= *numSquaresToMoveInCurrentDirection* <= 60
4. Each *Regular Protester* starts out with 5 hit-points.
5. Each *Regular Protester* starts out NOT in a *leave-the-oil-field* state.
6. Each *Regular Protester* have the following graphic parameters:
    a. It has an image depth of 0 – they're always in the foreground
    b. It has a size of 1.0

In addition to any other initialization that you decide to do in your *Regular Protester* class, a *Regular Protester* object must set itself to be visible using the *GraphObject* class's *setVisible()* method, e.g.:

```
setVisible(true);
```

What a Regular Protester Must Do During a Tick

Regular Protesters, unlike the player, don't necessarily get to take an action during every tick of the game (this is to make the game easier to play, since if *Regular Protesters* moved once every tick, they'd move much faster than the typical player can think and hit the keys on the keyboard). The *Regular Protester* must therefore compute a value indicating how often they're allowed to take an action (e.g., once every N ticks). This number of ticks (also known as "resting ticks") may be computed as follows:

int ticksToWaitBetweenMoves = max(0, 3 – current_level_number/4)

So if the value of *ticksToWaitBetweenMoves* was 3, then the *Regular Protester* must "rest" for 3 ticks and may perform its normal behavior every 4th tick. If the value is zero, then the *Regular Protester* does something during every tick of the game.

Here is what a *Regular Protester* must do during *every* tick:

1. The *Regular Protester* must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. If the *Regular Protester* is in a "rest state" during the current tick, it must do nothing other than update its resting tick count, and immediately return.
3. Otherwise, if the *Regular Protester* is in a *leave-the-oil-field* state (because their hit points reached zero due to being repeatedly squirted by the *Iceman's* squirt gun or bonked by a falling *Boulder*), then:
    a. If the *Regular Protester* is at its exit point (at location x=60, y=60) then it will immediately set its status to dead so that it will be removed from the game at the end of the current tick (by your *StudentWorld* class).
    b. Otherwise, the *Regular Protester* must move one square closer to its exit point (at x=60, y=60). It must determine what direction to move based on a queue-based maze-searching algorithm like the one we learned in class. The *Regular Protester* may not walk through *Ice* or *Boulders* to reach its exit point (meaning that none of the 16 squares that comprise the 4x4 *Regular Protester* icon may overlap with a *Ice* object and that the *Regular Protester* must not come within (within means <=) a distance of 3.0 units of any *Boulders*). Hint: It's inefficient for every single *Regular Protester* to try to figure out its own optimal path to the exit, so see if you can figure out a way to compute a single data structure that enables all *Protesters* to quickly determine their optimal path to their common exit point of 60,60, no matter where they are located in the oil field.
    c. After moving one step toward their exit point, the *Regular Protester* will return immediately. It will not perform any of the following steps.
4. Otherwise, the *Regular Protester* will check to see if they are within a distance of 4 units of the *Iceman*, AND they are currently facing in the *Iceman*'s direction[2]. If both are true and the *Regular Protester* hasn't shouted within its last non-resting 15 ticks[3], then the *Regular Protester* will:
    a. Play the shouting sound: SOUND_PROTESTER_YELL
    b. Inform the *Iceman* that he's been annoyed for a total of 2 annoyance points (deducting 2 points from the *Iceman's* hit points, and possibly

[2] For example, if the *Iceman* is at location x=0,y=10 and the *Regular Protester* is facing left at location x=4, with a 6<=y<=14, then the *Regular Protester* would be facing the *Iceman*. Similarly, *a Regular Protester* at x=3, y=9 facing *up* would also be facing the *Iceman*.

[3] A non-resting tick is one in which the *Protester* was not resting, and its *doSomething()* method actually was able to perform a substantive activity like moving, shouting, etc.

causing the *Iceman* to become sufficiently annoyed to quit the level (aka die)).
    c. Update some state variable in a manner that prevents this *Regular Protester* from shouting again for at least 15 non-resting ticks.
    d. Return immediately.
5. Otherwise, if the *Regular Protester*:
    a. Is in a straight horizontal or vertical line of sight to the *Iceman* (even if the *Regular Protester* isn't currently facing the *Iceman*), and
    b. Is more than 4 units away from the *Iceman* – that is, the radius from the *Regular Protester* and the *Iceman* is greater than 4.0 units away, and
    c. Could actually move the entire way to the *Iceman* with no *Ice* or *Boulders* blocking its path[4] (assuming it kept walking straight over the next N turns),

    Then the *Regular Protester* will:
    a. Change its direction to face in the direction of the *Iceman,* AND then take one step toward him.
    b. The *Regular Protester* will set its *numSquaresToMoveInCurrentDirection* value to zero, forcing it to pick a new direction/distance to move during its next non-resting tick (unless of course, the *Regular Protester* still sees the *Iceman* in its line of sight, in which case it will continue to move toward the *Iceman*).
    c. Then the *Regular Protester* will immediately return.
6. Otherwise, the *Regular Protester* can't directly see the *Iceman.* As such, it will decrement its *numSquaresToMoveInCurrentDirection* variable by one. If the *Regular Protester* has finished walking *numSquaresToMoveInCurrentDirection* steps in its currently-selected direction (i.e., *numSquaresToMoveInCurrentDirection <= 0),* then:
    a. The *Regular Protester* will pick a random new direction to move (up, down, left or right).
    b. If the random direction is blocked either by *Ice* or a *Boulder* such that it can't take even a single step in that chosen direction, then it will select a different direction and check it for blockage (it will continue checking directions until it has picked a direction that is not blocked).
    c. The *Regular Protester* will then change its direction to this new chosen direction.
    d. The *Regular Protester* will then pick a new value for *numSquaresToMoveInCurrentDirection* that will govern how far it should move in the selected direction over the next *numSquaresToMoveInCurrentDirection* non-resting ticks. The value must be such that: 8 <= *numSquaresToMoveInCurrentDirection* <= 60.
    e. The *Regular Protester* will then continue with step 8.
7. Otherwise, if the *Regular Protester*:
    a. Is sitting at an intersection where it could turn and move at least one square in a perpendicular direction from its currently facing direction (e.g., it is

[4] A *Regular Protester* cannot move to a location that is within a radius of 3 (<= 3.0) units of a *Boulder*.

currently facing left, and is at a junction where it could turn and move one step either upward/downward without being blocked by *Ice* or a *Boulder*), and

b. The *Regular Protester* hasn't made a perpendicular turn in the last 200 non-resting ticks.

Then the *Regular Protester* will:

a. Determine which of the two perpendicular directions are viable (a viable direction is one that allows movement of at least one square without the *Regular Protester* being blocked by *Ice* or a *Boulder*).
b. Pick a viable perpendicular direction. If both perpendicular directions are viable, then pick one of the two choices randomly.
c. Set its direction to the selected perpendicular direction.
d. Pick a new value for *numSquaresToMoveInCurrentDirection* that will govern how far it should move in the selected perpendicular direction over the next *numSquaresToMoveInCurrentDirection* non-resting ticks. The value must be such that: 8 <= *numSquaresToMoveInCurrentDirection* <= 60.
e. Remember that it made a perpendicular turn so that it doesn't make another perpendicular turn for at least 200 more non-resting ticks.
f. Continue on with the next step.

8. Finally, the *Regular Protester* will then attempt to take one step in its currently facing direction (which, by the way, might have just been changed by one of the last few steps above).
9. If the *Regular Protester* is for some reason blocked from taking a step in its currently facing direction, it will set the *numSquaresToMoveInCurrentDirection* to zero, resulting in a new direction being chosen during the *Regular Protester's* next non-resting tick (but not the current tick – the *Regular Protester* must do nothing during the current tick).

Note that only *Ice* and *Boulders* block a *Regular Protester* from moving. *Regular Protesters* can occupy the same square as all other game objects including the *Iceman* and other *Protesters*.

What the *Regular Protester* Must Do When It Is Annoyed

A *Regular Protester* can't be further annoyed once it is in a *leave-the-oil-field* state – this means it can't be squirted or bonked by *Boulders* while in this state.

When the *Regular Protester* is annoyed (e.g., by being hit by a *Squirt* object fired by the *Iceman* or being bonked by a *Boulder*), its hit-points should be decremented by the appropriate amount (the amount is specified elsewhere in this document).

If, after its hit-points have been decremented, the *Regular Protester* hasn't been completely annoyed (to the point it wants to leave the oil field) then it must play a sound of: SOUND_PROTESTER_ANNOYED. It will then be "stunned" and placed in a resting state for N resting ticks, where:

N = max(50, 100 – current_level_number * 10)

Note: This essentially extends the *Regular Protester's* normal resting state to N ticks. Remember, that when in a resting state, the *Regular Protester* will immediately return when its *doSomething()* method is called.

On the other hand, if the *Regular Protester's* hit-points reach zero or below due to being annoyed for any reason, the *Regular Protester*:

1. Transitions into a *leave-the-oil-field* state.
2. Plays an "I give up" sound effect: SOUND_PROTESTER_GIVE_UP.
3. Sets its resting tick count to zero, ensuring that it will do something on the very next game tick, even if it was previously stunned or otherwise in a resting state.
4. If the *Regular Protester* was annoyed due to being bonked by a *Boulder*, then it will increase the player's score by 500 points.
5. On the other hand, if the *Regular Protester* was annoyed due to being squirted repeatedly, then it will increase the player's score by 100 points.

What the *Regular Protester* Must Do When It Picks up a *Gold Nugget*

When a *Gold Nugget* has activated due to being stepped upon by a *Regular Protester,* it will notify the *Regular Protester* that it has picked up a *Nugget*. The *Regular Protester* must do the following in response to being notified that it has just stepped upon a *Nugget*:

1. The *Regular Protester* plays an "I'm rich" sound effect: SOUND_PROTESTER_FOUND_GOLD.
2. The *Regular Protester* increases the player's score by 25 points for the bribery.
3. The *Regular Protester* will immediately be bribed and transition into a *leave-the-oil-field* state.

Hardcore Protesters

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

What a Hardcore Protester object Must Do When It Is Created

When it is first created:

1. Each *Hardcore Protester* must have an image ID of IMID_HARD_CORE_PROTESTER.
2. Each *Hardcore Protester* must start out facing left.

3. Each *Hardcore Protester* must decide how many squares, *numSquaresToMoveInCurrentDirection*, it will decide to move left before possibly switching its direction. This value must be: 8 <= *numSquaresToMoveInCurrentDirection* <= 60
4. Each *Hardcore Protester* starts out with 20 hit-points.
5. Each *Hardcore Protester* starts out NOT in a *leave-the-oil-field* state.
6. Each *Hardcore Protester* have the following graphic parameters:
    a. It has an image depth of 0 – they're always in the foreground
    b. It has a size of 1.0

In addition to any other initialization that you decide to do in your *Hardcore Protesters* class, a *Hardcore Protester* object must set itself to be visible using the *GraphObject* class's *setVisible()* method, e.g.:

```
setVisible(true);
```

What a Hardcore Protester Must Do During a Tick

Hardcore Protesters, unlike the player, don't necessarily get to take an action during every tick of the game (this is to make the game easier to play, since if *Hardcore Protesters* moved once every tick, they'd move much faster than the typical user can think and hit the keys on the keyboard). The *Hardcore Protester* must therefore compute a value indicating how often they're allowed to take an action (e.g., once every N ticks). This number of ticks (also known as "resting ticks") may be computed as follows:

int ticksToWaitBetweenMoves = max(0, 3 – current_level_number/4)

So if the value of *ticksToWaitBetweenMoves* was 3, then the *Hardcore Protester* must "rest" for 3 ticks and may perform its normal behavior every 4 th tick. If the value is zero, then the *Hardcore Protester* does something during every tick of the game.

Here is what a *Hardcore Protester* must do during *every* tick:

1. The *Hardcore Protester* must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. If the *Hardcore Protester* is in a "rest state" during the current tick, it must do nothing other than update its resting tick count, and immediately return.
3. Otherwise, if the *Hardcore Protester* is in a *leave-the-oil-field* state (because their hit points reached zero due to being repeatedly squirted by *Squirt* objects fired from the *Iceman's* squirt gun or bonked by a falling *Boulder*), then:
    a. If the *Hardcore Protester* is at its exit point (at location x=60, y=60) then it will immediately set its status to dead so that it will be removed from the game at the end of the current tick (by your *StudentWorld* class).
    b. Otherwise, the *Hardcore Protester* must move one square closer to its exit point (at x=60, y=60). It must determine what direction to move based on a

queue-based maze-searching algorithm like the one we learned in class. The *Hardcore Protester* may not walk through *Ice* or *Boulders* to reach its exit point (meaning that none of the 16 squares that comprise the 4x4 *Hardcore Protester* icon may overlap with *Ice* and that the *Hardcore Protester* must not come within (within means <=) a distance of 3.0 units of any *Boulders*). Hint: It's inefficient for every single *Hardcore Protester* to try to figure out its own optimal path to their common exit point of 60,60, so see if you can figure out a way to compute a single data structure that enables all *Protesters* to quickly determine their optimal path to their common exit, no matter where they are located in the oil field.

   c. After moving one step toward their exit point, the *Hardcore Protester* will return immediately. It will not perform any of the following steps.

4. Otherwise, the *Hardcore Protester* will check to see if they are within a distance of 4 units of the *Iceman*, AND they are currently facing in the *Iceman*'s direction[5]. If both are true and the *Hardcore Protester* hasn't shouted within the last non-resting 15 ticks, then the Hardcore Protester will:
   a. Play the shouting sound: SOUND_PROTESTER_YELL
   b. Inform the *Iceman* that he's been annoyed for a total of 2 annoyance points (deducting 2 points from the *Iceman's* hit points, and possibly causing the *Iceman* to become sufficiently annoyed to quit the level (aka die)).
   c. Update some state variable in a manner that prevents this *Hardcore Protester* from shouting again for at least 15 non-resting ticks.
   d. Return immediately.
5. Otherwise, if the *Hardcore Protester* is more than 4.0 units away from the *Iceman*, the *Hardcore Protester* will:
   a. Compute a value M, such that M = 16 + current_level_number * 2
   b. If the *Hardcore Protester* is less than or equal to a total of M legal horizontal or vertical moves away from the current location of the *Iceman* (as can be determined using the same basic queue-based maze-searching algorithm described in *Hardcore Protester* item 3.b above), then the *Hardcore Protester* will sense the *Iceman*'s cell phone signal and know where he is. This is true even if the *Hardcore Protester* has no direct line of sight to the *Iceman*, so long as he can be reached in M moves without digging through *Ice* or going through a *Boulder*. In such a situation, the *Hardcore Protester* will:
      i. Determine which horizontal/vertical direction to move in (as dictated by its maze-searching algorithm) such that if it were to make such a move it would be one square closer to the *Iceman*.
      ii. Change its current direction to face this new direction.
      iii. Move one square in this direction such that after the move it is one square closer to the *Iceman*'*s* current location.

[5] For example, if the *Iceman* is at location x=0,y=10 and the *Regular Protester* is facing left at location x=4, with a 6<=y<=14, then the *Regular Protester* would be facing the *Iceman*. Similarly, *a Regular Protester* at x=3, y=9 facing *up* would also be facing the *Iceman*.

iv. Return immediately.

6. Otherwise, if the *Hardcore Protester*:
    a. Is in a straight horizontal or vertical line of sight to the *Iceman* (even if the *Hardcore Protester* isn't currently facing the *Iceman*), and
    b. Is more than 4 units away from the Iceman – that is the radius from the *Hardcore Protester* and the *Iceman* is greater than 4.0 units away, and
    c. Could actually move the entire way to the *Iceman* with no *Ice* or *Boulders* blocking its path[6] (assuming it kept walking straight over the next N turns),

    Then the *Hardcore Protester* will:
    d. Turn to face the *Iceman* and take one step toward him.
    e. The *Hardcore Protester* will set its *numSquaresToMoveInCurrentDirection* value to zero, forcing it to pick a new direction/distance to move during its next non-resting tick (unless of course, the *Hardcore Protester* still sees the *Iceman* in its line of sight).
    f. Then the *Hardcore Protester* will immediately return.
7. Otherwise, the *Hardcore Protester* will decrement its *numSquaresToMoveInCurrentDirection* variable by one. If the *Hardcore Protester* has finished walking its *numSquaresToMoveInCurrentDirection* steps in its currently-selected direction (i.e., *numSquaresToMoveInCurrentDirection <= 0),* then:
    a. The *Hardcore Protester* will pick a random new direction to move in (up, down, left or right).
    b. If the random direction is blocked either by *Ice* or a *Boulder* such that it can't take even a single step in that chosen direction, then it will select a different direction and check it for blockage (it will continue checking directions until it has picked a direction that is not blocked).
    c. The *Hardcore Protester* will then change its direction to this new chosen direction.
    d. The *Hardcore Protester* will then pick a new value for *numSquaresToMoveInCurrentDirection* that will govern how far it should move in the selected direction over the next *numSquaresToMoveInCurrentDirection* non-resting ticks. The value must be such that: 8 <= *numSquaresToMoveInCurrentDirection* <= 60.
    e. The *Hardcore Protester* will then continue with step 8.
8. Otherwise, if the *Hardcore Protester*:
    a. Is sitting at an intersection where it could turn and move at least one square in a perpendicular direction from its currently facing direction (e.g., it is currently facing left, and is at a junction where it could turn and move one step either upward/downward without being blocked by *Ice* or a *Boulder*), and
    b. The *Hardcore Protester* hasn't made a perpendicular turn in the last 200 non-resting ticks.

    Then the *Hardcore Protester* will:

[6] A *Regular Protester* cannot move to a location that is within a radius of 3 (<= 3.0) units of a *Boulder*.

c. Determine which of the two perpendicular directions are viable (a viable direction is one that allows movement of at least one square without the *Hardcore Protester* being blocked by *Ice* or a *Boulder*).
d. Pick a viable perpendicular direction. If both perpendicular directions are viable, then pick one of the two choices randomly.
e. Set its direction to the selected perpendicular direction.
f. Pick a new value for *numSquaresToMoveInCurrentDirection* that will govern how far it should move in the selected perpendicular direction over the next *numSquaresToMoveInCurrentDirection* non-resting ticks. The value must be such that: 8 <= *numSquaresToMoveInCurrentDirection* <= 60.
g. Remember that it made a perpendicular turn so that it doesn't make another perpendicular turn for at least 200 more non-resting ticks.
h. Continue on with the next step.

9. Finally, the *Hardcore Protester* will then attempt to take one step in its currently facing direction (which, by the way, might have just been changed by one of the last few steps above).
10. If the *Hardcore Protester* is for some reason blocked from taking a step in its currently facing direction, it will set the *numSquaresToMoveInCurrentDirection* to zero, resulting in a new direction being chosen during the *Hardcore Protester's* next non-resting tick (but not the current tick – the *Hardcore Protester* must do nothing during the current tick).

Note that only *Ice* and *Boulders* block a *Hardcore Protester* from moving. *Hardcore Protesters* can occupy the same square as all other game objects including the *Iceman* and other *Protesters*.

What the *Hardcore Protester* Must Do When It Is Annoyed

A *Hardcore Protester* can't be further annoyed once it is in a *leave-the-oil-field* state – this means it can't be squirted or bonked by *Boulders* while in this state.

When the *Hardcore Protester* is annoyed (e.g., by being squirted by the *Iceman* or being bonked by a *Boulder*), its hit-points should be decremented by the appropriate amount (the amount is specified elsewhere in this document).

If, after its hit-points have been decremented, the *Hardcore Protester* hasn't been completely annoyed (to the point it wants to leave the oil field) then it must play a sound of: SOUND_PROTESTER_ANNOYED. It will then be "stunned" and placed in a resting state for N resting ticks, where:

N = max(50, 100 – current_level_number * 10)

Note: This essentially extends the *Hardcore Protester's* normal resting state to N ticks. Remember, that when in a resting state, the *Hardcore Protester* will immediately return when its *doSomething()* method is called.

On the other hand, if the *Hardcore Protester's* hit-points reach zero or below due to being annoyed for any reason, the *Hardcore Protester*:

1. Transitions into a *leave-the-oil-field* state.
2. Plays an "I give up" sound effect: SOUND_PROTESTER_GIVE_UP.
3. Sets its resting tick count to zero, ensuring that it will do something on the very next game tick, even if it was previously stunned or otherwise in a resting state.
4. If the *Hardcore Protester* was annoyed due to being bonked by a *Boulder*, then it will increase the player's score by 500 points.
5. On the other hand, if the *Hardcore Protester* was annoyed into submission due to being squirted repeatedly, then it will increase the player's score by 250 points.

What the *Hardcore Protester* Must Do When It Picks up a *Gold Nugget*

When a *Gold Nugget* has activated due to being stepped upon by a *Hardcore Protester,* it will notify the *Hardcore Protester* that it has picked up a *Nugget*. The *Hardcore Protester* must do the following in response to being notified that it has just stepped upon a *Nugget*:

1. The *Hardcore Protester* plays an "I'm rich!" sound effect: SOUND_PROTESTER_FOUND_GOLD.
2. The *Hardcore Protester* increases the player's score by 50 points for the bribery.
3. The *Hardcore Protester* will become fixated on the *Nugget* and will pause to stare at it (just as if he/she were in a resting state – doing nothing else) for the following number of game ticks:

   ticks_to_stare = max(50, 100 – current_level_number * 10)

4. After the specified number of game ticks, the *Hardcore Protester* will become bored with the *Nugget* and continue to pursue the *Iceman* using its normal algorithm, as described above.

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 always fail to solve this project, 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, 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.

Building the Game

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

To build the game, follow these steps:

For Windows

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

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

by double-clicking on the executable file, the Assets folder should be in the same folder as the executable.

For Mac OS X

Unzip the Iceman-skeleton-mac.zip archive into a folder on your hard drive.
Double-click on our provided Iceman.xcodeproj to start Xcode.
If you build and run your program from within Xcode, the Assets directory should be in the directory yourProjectDir*/DerivedData/*yourProjectName*/BuildProducts/Debug* (e.g., */Users/fred/Iceman/DerivedData/Iceman/Build/Products/Debug*). On the other hand, if you launch the program by double-clicking on the executable file, the Assets directory should be in your home directory (e.g., */Users/fred*).

What to Turn In

Part #1

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 2, your job is to build a really simple version of the Iceman 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 objects (e.g., the *Iceman*, *Regular Protesters*, *Hardcore Protesters*, *Barrels* of oil, *Nuggets*, *Ice*, etc.):
    i. It must have a simple constructor and destructor.
    ii. It must be derived from our *GraphObject* class.
    iii. It (or its base class) must make itself visible via a call to *setVisible(true);*
    iv. It must have a virtual method called *doSomething()* that can be called by the World to get one of the game's actors to do something.
    v. You may add other public/private methods and private member variables to this base class, as you see fit.
2. A *Ice* class, derived in some way from the base class described in 1 above:
    i. It must have a simple constructor and destructor that initialize a new *Ice* object.
    ii. It must have an Image ID of IMID_ICE.
    iii. You may add any set of public/private methods and private member variables to your *Ice* class as you see fit, so long as you

use good object oriented programming style (e.g., you must not duplicate functionality across classes).

3. A limited version of your *Iceman* class, derived in some way from the base class described in 1 just above (either directly derived from the base class, or derived from some other class that is somehow derived from the base class):
    i. It must have a simple constructor and destructor that initializes/deinitializes the *Iceman* – see the *Iceman* section for more details on where to initially place the *Iceman* on the screen.
    ii. It must have an Image ID of IMID_PLAYER.
    iii. It (or its base class) must make itself visible via a call to *setVisible(true);*
    iv. It must have a limited version of a *doSomething()* method that lets the user pick a direction by hitting a directional key. If the player hits a directional key during the current tick, it updates the *Iceman's* location to the target square, removing any *Ice* objects that overlap with the *Iceman's* 4x4 graphic image. To move the *Iceman*, all this *doSomething()* method has to do is properly adjust the player's X,Y coordinates using *GraphObject's moveTo()* method and our graphics system will automatically animate its movement it around the oil field!
    v. You may add any public/private methods and private member variables to your player class as you see fit, so long as you use good object oriented programming style (e.g., you must not duplicate functionality across classes).

4. Create a limited version of the *StudentWorld* class.
    i. Add any private member variables to this class required to keep track of all *Ice* in the oil field as well as the *Iceman* object. You may ignore all other items in the oil field such as *Boulders*, *Barrels of oil, Protesters*, *Nuggets*, etc. for part #1.
    ii. Implement a constructor for this class that initializes all member variables required for proper gameplay.
    iii. Implement a destructor for this class that frees any remaining dynamically allocated data that has not yet been freed at the time the class is destroyed (e.g., the *Iceman* and all remaining *Ice*).
    iv. Implement the *init()* method in this class. It must:
        1. Create the *Iceman* object and insert it into the oil field at the right starting location (see the *StudentWorld init()* section of this document for details on the starting location).
        2. Creates all of the oil field's *Ice objects* and inserts them into a data structure that tracks active Ice (see the *StudentWorld init()* section for details on where to place *Ice*, and what data structure to use track all of the remaining *Ice* in the game).
    v. Implement the *move()* method in your *StudentWorld* class. During each tick, it must ask your *Iceman* object to do something. Your *move()* method need not check to see if the *Iceman* has died or not; you may assume at this point that

the *Iceman* cannot die. Nor need your *move()* method deal with any *Protesters* or other actors (e.g., *Nuggets* or *Boulders*) at this point – just the *Iceman*.

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

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.

You'll know you're done with part 1 when your program builds and does the following: When it runs and the user hits Enter to begin playing, it displays a oil field filled with *Ice* (except for a single open vertical shaft dug down the middle of the oil field) with the *Iceman* in its proper starting position in the top-middle of the screen. If your base class(es) and *Iceman* class work properly, you should be able to move the *Iceman* around the oil field and dig through the *Ice* using the directional keys.

Note, the Part #1 specification above doesn't require you to implement any *Regular Protesters*, *Hardcore Protesters*, *Boulders*, *Barrels*, etc. (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 additonal functionality, make sure that your *Iceman*, *Ice*, and *StudentWorld* classes still work properly and that your program still builds and runs, and meets the requirements stated above for Part #1!**

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

What to Turn In For Part #1

You must turn in your source code for the simple version of your game, which **builds without errors** under either Visual Studio or Xcode. You do **not** have to get it to run under more than one compiler. You will turn in a zip file containing nothing more than these four files:

```
Actor.h            // contains base, Ice, and Iceman class declarations
                   // as well as constants required by these classes
Actor.cpp          // contains the implementations 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!

Part #2

After you have turned in your work for Part #1 of Project 2, 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 Iceman 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 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.doc, report.docx, or report.txt // your report
```

You must turn in a report that contains 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 class because all Actors in Iceman must have a blah function, and each type of actor defines their own special 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 the Squirt class." or "My Hardcore Protester doesn't work correctly yet so I just treat it like a Regular Protester 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 seems reasonable 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 builds! If we can sort of play your game, but it's not perfect, that's better than it not even building!

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 anyone.

GOOD LUCK!